The Rhetorical Knowledge Representation System Reference Manual (for Rhet Version 17.9)

Bradford W. Miller

Technical Report 326
November 1990

UNIVERSITY OF
ROCHESTER
COMPUTER SCIENCE

The Rhetorical Knowledge Representation System Reference Manual (For Rhet Version 17.9)

Bradford W. Miller

The University of Rochester
Computer Science Department
Rochester, New York 14627

Technical Report 326

November 1990

This work was supported in part by ONR research contract no. N00014-80-C-0197, in part by U.S. Army Engineering Topographic Laboratories research contract no. DACA76-85-C-0001, and in part by the Air Force Systems Command, Rome Air Development Center, Griffiss Air Force Base, New York 13441-5700, and the Air Force Office of Scientific Research, Bolling AFB, DC 20332 under Contract No. F30602-85-C-0008.

Abstract

Rhetorical (Rhet) is a programming / knowledge representation system that offers a set of tools for building automated reasoning systems. It's emphasis is on flexibility of representation, allowing the user to decide if the system will basically operate as a theorem prover, a frame-like system, or an associative network. Rhet may be used as the back-end to a user's programming system and handle the knowledge representation chores, or it may be used as a full-blown programming language.

Rhet offers two major modes of inference: a horn clause theorem prover (backwards chaining mechanism), and a forward chaining mechanism. Both modes use a common representation of facts, namely horn clauses with universally quantified, potentially type restricted, variables, and use the unification algorithm. Additionally, they both share the following additional specialized reasoning capabilities:

1. variables may be typed with a fairly general type theory that allows a limited calculus of types including intersection and subtraction;
2. full reasoning about equality between ground terms;
3. reasoning within a context space, with access to axioms and terms in parent contexts;
4. escapes into Lisp for use as necessary.

Contents

List of Tables

Chapter 1

Introduction

1.1 What is this?

This documents the user interface for the Rhetorical[1] (Rhet) system. This system, functionally, is an extension to the HORNE[Allen and Miller, 1986] system we had been using at the University of Rochester, and has been leveraged off of our experience with that system.

Rhet is a horn clause based reasoning system that is embedded in a Common Lisp (CL) [Steele Jr., 1990] environment[2]. Its facilities are called as Lisp functions and Rhet programs can themselves call Lisp functions. Thus, effective programming in Rhet involves a careful mixture of logic programming and Lisp programming. This manual assumes that the user is familiar with the fundamentals of Rhet as described in [Allen and Miller, 1990].

In some sense, there are two user interfaces that are supported. The simpler one is the programmatic user interface: that is, the functions that are supplied for a programmer to gain access to Rhet. From such an interface, a user can expect to be able to do proofs, add facts and axioms but not, perhaps, reorder axioms or debug much in the way of Rhet problems. The user who utilizes Rhet's more screen oriented tools for editing axioms, doing queries and making assertions will garner a definite benefit for debugging his system. It is possible, of course, to have the best of both worlds: namely, use the screen oriented utilities to get up that part of your system you expect Rhet to do, and then tie your own software in via the more primitive interface. This choice, is left to the user, however.

The following sections define the Rhet language as supported by the interface, the programmatic interface, and describes the enhanced screen-oriented interface[3]. Notes in the margin point out the significant departures from the version 15.25manual. A certain

[1] Rochester Horn clause Extended Tool Of Research In Characterizing Applied natural Language

[2] Some manufacturer defined extensions are used, but they are minimal, for ease of rehosting between lispmachine type environments.

[3] For the most part, the screen oriented interface functionally builds on the existing facilities, such as ZMACS, of the Explorer™ and Symbolics™ environments. It is not expected to be particularly portable to non-Lispms, unlike the rest of the system. This is not to say that the rest of the system will not require porting, but for the most part, all machine dependent code is isolated.

minimal amount of familiarity with Lisp, and Prolog, is assumed[4]. For further internal details about Rhet, see [Miller, 1990b].

A small initial note on notation used herein: Throughout we use "()"s to denote Lisp lists, in a manner familiar to Lisp users. We use "[]"s to denote Rhet objects, such as axioms, expressions. or facts. In fact, the Rhet program expects these symbols in it's input as well, to disambiguate between Rhet and Lisp objects, as described in detail in section 2.1. A rhet object that appears alone, e.g.

```
(1.1)    [[F ?x] <foo [P ?x] [Q ?x]]
```

is considered to be an assertion, that is, we are adding the above statement to the KB. Briefly, in the above statement, a horn clause or an axiom, the clause to the left of "<foo" is the left hand side *form* of the axiom, usually abbreviated as the *LHS*, and the clauses to the right of "<foo" are considered the right hand side forms, usually abbreviated as the *RHS*. The token "<foo" itself is referred to as the index of the axiom. The capitol letters are constants, and in this case are all predicates (they occupy the first position in a clause) and the symbols beginning with a question mark are all variables.

The above is distinguished from, for example,

```
(1.2)    ((F ?x) <foo (P ?x) (Q ?x))
```

which is not an axiom at all, but a lisp list with four elements, a list, an atom, and two more lists. Note that even in this example, however, the symbol ?x is still a Rhet variable, even though the capitol letters are all lisp atoms.

1.2 Acknowledgments

Special acknowledgment is given to Michael McInerny, who has invested a considerable amount of RA and programming time into the type subsystem and user contexts; Stephane Guez for his work on structured types; Jun Tarui for his work on TMS and function typing; Nat Martin for his RA work on the parser; and Steven Feist, for his RA work on the type subsystem. I'd also like to acknowledge all the early users of Rhet who suffered through the MTBF of about twenty minutes to help us achieve the current[5] version, and in particular Eliz Hinkelman and Keri Jackson. The TEMPOS reasoning facilities are due to the extraordinary efforts of Hans Koomen, whose constructive criticism allowed much better interfaces for user constructed extensions and builtins to be effected. And of course, the project was conceived of and continues to be guided by James Allen.

[4] And to a small extent Lisp machines.

[5] much more stable ⌣

1.3 Help Us!

We want to know about problems you have using Rhet, inconsistencies with the manual, or suggestions on how any part of Rhet might be improved (*e.g.* to make it easier to use, the manual more clear, the index more functional, *etc.*). Send mail to one of the rhet discussion lists: Rhet@cs.rochester.edu or Bug-Rhet@cs.rochester.edu depending on whether you are asking a question, or proposing an enhancement (the former list is appropriate), or reporting a bug or inconsistency. You may also contact the author directly using the address given on the title page of this document.

Chapter 2

The Language

2.1 Conventions

This document (and Rhet) follows the following basic conventions. Most Rhet expressions appear inside of square brackets, *i.e.* "[" and "]" to distinguish them from Lisp lists, which use the normal parenthesis notation. Inside of a Rhet expression, anything which looks like a normal Lisp atom is actually a *Rhet term. Real Lisp atoms, should they be needed*, are preceded by a colon (:), unless they appear inside of a Lisp list, in which case normal Lisp syntax prevails. Rhet variables always have a question mark (?) as their first character[1]. Thus

(2.1) `[P ?x (A B [C]) :D]`

is a Rhet expression whose head (or Predicate Name following the usage in table 2.1) is the Rhet term P; and whose arguments are, in order, a Rhet variable, a Lisp list (consisting of two Lisp atoms and a Rhet Predicate) and a Lisp atom. Functions are documented along the lines of the standard Lisp reference literature, *e.g.* [Steele Jr., 1990].

2.2 Syntax

The six major classes of expressions in this language are Terms, Predicates, Forms, Facts, Function Terms and Axioms. The syntax for these classes are given by the BNF rules in table 2.1. Note that in this table quotes ("") delimit an actual string or token expected by the reader (thus the rule for a Lisp-Atom indicates that an actual typed-in literal colon (:) must be followed by a constant); "|" is an or sign, "*" represents Kleene Closure, and "+" is like Kleene Closure, but requires at least one occurrence. Mod

A Fact is any Rhet expression that is asserted as true and does not contain variables or a RHS. Thus,

[1]with the exception of don't care variables.

(2.2) `[P A]`

is potentially a Fact (it could be asserted), as is

(2.3) `[P :F (Foo Bar)]`

and

(2.4) `[[P B] <]`

is potentially a Fact (if asserted – the presence of an index removes ambiguity with a Form *q.v.*) but neither

(2.5) `[P ?x]`

nor

(2.6) `[[P C] < [P D]]`

are since the former includes a variable, and the latter has a RHS (symbols beginning with '<' are considered an index). Forms include Facts, but also include Rhet expressions with variables. Forms cannot have an index. Thus

(2.7) `[P ?x]`

is a Form, as is

(2.8) `[Q [P ?x]]`,

but

(2.9) `[[P ?z] <]`

isn't since it includes an index. Expressions that include an index and are not a Fact (that is, they either have variables, or a RHS) are Axioms.

(2.10) `[[P ?x] <]`

is an axiom, as is

(2.11) `[[P C] < [P D]]`.

Lisp expressions may be involved in Forms, Axioms or Facts, but never in Function Terms.

(2.12) `[P :A]`

is a Form, or a Fact, but

(2.13) `[P [C (D E)]]`

may only be a Form, since the subexpression is not a Function Term. Note that Function Terms are syntactically indistinguishable from certain Facts, the difference being that instead of a Predicate Name in the first position of the expression, the name of a Builtin function or declared Lisp function will be used[2].

Rhet also accepts numeric constants as well as literal atoms as constants. In particular, Common Lisp floating point numbers, rationals, and integers may each or all be used as part of a term. Note that Function-Terms may not consist of numeric-constants in whole or part. New

Note that what is considered to be an atom is the same as a keyword in Common Lisp. Further, all atomic (non-string) terms are converted internally to upper-case, just as in Common Lisp. For example,

(2.14) `[P-x a-b-c ?x*foo]`

and

(2.15) `[p-x A-B-C ?x*FOO]`

are internally represented identically. Normally, output is in upper case only, with the exception of variables which do not force case (that is, ?x is distinct from ?X). This document will follow the convention of using upper-case names for atoms, and lower-case for variables (following the '?' symbol). Indexes, like variables, are considered to be strings, and therefore case is preserved. Thus, "<foo" is not the same index as "<Foo". All indexes are required to begin with the character '<' to distinguish them from forms when parsing an axiom vs. a form. Types will always be prefixed with "T-" to conform to the naming regularities expected of structured types. Mod

An example of an axiom is:

(2.16) `[[P ?x] <1 [Q ?x]]`

where "`[P ?x]`" is the conclusion, "<1" is the index, and "`[Q ?x]`" is a simple predicate. This statement is interpreted as follows: the assertion named[3] "<1" signifies that for any x, `[Q x]` implies `[P x]`. Or, alternately, to prove the predicate `[P x]`, for any x, proof of the predicate `[Q x]` suffices. Mod

An example of a set is:

(2.17) `{[A] (B C) [[Foo ?x] < [Bar ?x]]}`

which would be a set consisting of a Form, a List, and an Axiom.

---

[2]Because of this ambiguity, it is possible to confuse Rhet by using a term as a function term, then later asserting it. In the semantics of CLtL ([Steele Jr., 1990]) this is an error.

[3]Actually, 'named' may be a misnomer, since the index can be non-unique.

```
<Set>             ::=  "{" <<Axiom> | <Term>>* "}"

<Orthodox Set>    ::=  "{" <<Function Term>>* "}"

<Fact>            ::=  "[" <Predicate Name> <<Lisp Expression> |
                            <Function Term>>* "]"

<Indexed-Fact>    ::=  "[" <Predicate Name>
                            <<Lisp Expression> | <Function Term>>*
                            <Index> "]" |
                       "[" <Fact> <Index> "]"

<Function Term>   ::=  "[" <Function Name> <Function Term>* "]" |
                       <Constant> |
                       <Orthodox Set>

<Axiom>           ::=  <FC Axiom> | <BC Axiom>

<FC Axiom>        ::=  "[" <Predicate> <Index> <Predicate>*
                         ":Forward" <<Predicate>* | ":All" > "]"

<BC Axiom>        ::=  "[" <Predicate> <Index> <Predicate>* "]"

<Predicate>       ::=  "[" <Predicate Name>
                            <<Function Term> | <Form>>* "]" |
                       <Fact>
                       "[" NOT <Predicate Name>
                            <<Function Term> | <Form>>* "]" |
                       "[" NOT <Predicate> "]"

<Form>            ::=  <Function Term> | <Predicate> |
                       "[" <<Term> | <Form>>+ "]"

<Index>           ::=  "<" <String-Constant>
```

Table 2.1: Syntax Table — Part I

<Predicate Name>	::=	<Constant>
<Function Name>	::=	<Constant>
<Term>	::=	<Constant'> | <Variable> | <List> | <Predicate> | <Lisp-Atom> | <Function Term>
<Lisp Expression>	::=	<Lisp-Atom> | <Ground List>
<List>	::=	“(” <Term>* “)” | “(” <Term>+ “.” <Term> “)” | “NIL”
<Ground List>	::=	“(” <Constant'>* “)” | “(” <Constant'>+ “.” (<Constant'> | <Ground List>) “)” | “NIL”
<Type-Expr>	::=	<Type> | “(” <Type>* “)” | “(” <Type>* - <Type>* “)”
<Type>	::=	“T-” <Constant>
<Lisp-Atom>	::=	“:” <Constant>
<Variable>	::=	“?” <Constant> | “?” <Constant> “*” < Type-Expr > | “[” “ANY” <Variable> <Predicate>+“]” | “_”
<Constant'>	::=	<Constant> | <Numeric Constant>

Table 2.2: Syntax Table — Part II

2.2.1 Special Symbols

The Rhet system uses several special symbols which should not be used for other purposes. They are defined to be reader macro characters, and loading Rhet may make destructive changes to the default readtable[4].

? in the printed representation indicates a variable. It will cause the atom following it to be expanded into the internal variable format on input.

New

_ in the printed representation indicates a don't care variable. This differs from the "?" type of variable in that it has no argument, no type, and two instances in the same term are always different objects. Don't care variables unify with anything.

***** in the printed representation indicates that the expression following it is a type. This is true only in axioms and constants, and normally must follow a variable. The symbol can be used freely in Lisp code.

xB operator in the printed representation indicates a belief operator. This is only true in axioms and constants. These symbols can be used freely in Lisp code without special interpretation. Examples of belief operators are "MB" and "SBHBMB".

; in source code is considered a comment start, and text between it and the end of the line is not interpreted.

New

#! converts a lisp atom or list to an (internal) type expression. This may be useful with certain builtins that can take a type as an argument.

#| in source code is the beginning of a structured comment. Everything between this and the end structured comment character is ignored. This is usually written as #|| since the Lisp reader treats |anything at all| as a single token, so in this manner a comment is treated as one token to be ignored.

|# in source code is the end of a structured comment. This is usually written as ||# as above.

[in source code is considered the beginning of some Rhet structure.

] in source code is considered the end of some Rhet structure.

#[in source code is considered the beginning of a Rhet binding environment. This is needed when more than one Rhet structure or variable will appear at top level, but the variables in these structures are meant to be the same if they print the same. For example, when doing a Prove-All where each clause may refer to the same variable ?x, as in:

```
(2.18)    (Prove-All #[[IS-STEP-IN-SOME-PLAN [C-GO ?X ?Y] ?Z*T-PLANS]
                       [PRINT-PLAN ?Z]#])
```

[4] This is an installation option, see appendix of [Miller, 1990b].

the #[#] forms make sure the references to ?z are the same in the two goals. Otherwise they would be assumed to be different, since they are otherwise independent clauses. In this case, we could also have used the [And] builtin to get around this problem, however examine the following:

(2.19)
```
#[ ; these force the read of the defrhetpred into a single
   ; variable binding environment.
(DEFRHETPRED IS-PLAN-STEP-LF
      (&BOUND ?STEP*T-ACTION ?PLAN*T-PLANS)
    "Like IS-PLAN-STEP, but a lispfunction, so in theory faster.
     Only works if ?step is not a form with variables!"
    (DECLARE (MONOTONIC))

    (MEMBER ?STEP (Relation-List :PLAN-STEPS ?PLAN)))
#] ; close the special reading environment.
```

Since this DefRhetPred is for a lispfn, we have to use these reader symbols to make sure the reference to the variable in the arglist is the same as the one in the embedded Rhet form. Had no Rhet forms been embedded, it would not have been needed: DefRhetPred is smart enough to handle unembedded references to arglist variables itself.

] End of a Rhet binding environment.

{ is a set constructor, and begins a sequence to be taken as denoting a set.

} ends a set constructor.

2.3 Typed Terms

The type of a variable is indicated by appending a suffix to the variable indicating its type. Thus ?x*CAT names a variable ?x that is of type *CAT. The variable ?x*CAT will unify only with terms that are compatible with the type *CAT. The type of a variable is by default *T-U, the most general type for Rhet objects. This cannot be bound to a Lisp object, however. Inside of a List, the default type of a Rhet variable is *T-Lisp which can only be bound to Lisp objects. Table 2.3 lists the default types Rhet predefines for the user. New

Types should be viewed as sets, and no restrictions are assumed as to whether sets are disjoint, mutually exclusive, or wholly contained by each other. This information is specified by the user by declaring the type hierarchy as defined in section 6. Instances of types may be specified with the following Lisp functions (which are also available as Builtins):

Typename	Description
T-Anything	The root of the type tree. All Rhet objects and terms are subtypes of this type.
T-U	The Universal Type. All Function Terms are subtypes of this type, as are any instances the user would normally create.
T-Set	The Set Type. All set objects are subtypes of this type. This type overlaps T-U.
T-Orthodox-Set	The named intersection between T-U and T-Set; Set objects that can be involved in equality are subtypes of this.
T-Fact	The type of all Facts. It is disjoint from all other subtypes of T-Anything.
T-Axiom	The type of all Axioms. It is disjoint from all other subtypes of T-Anything.
T-FC-Axiom	The type of Forward Chaining Axioms. It is a subtype of T-Axiom, disjoint from T-BC-Axiom.
T-BC-Axiom	The type of Backward Chaining Axioms. It is a subtype of T-Axiom, disjoint from T-FC-Axiom.
T-Type	The type of all Type objects. It is disjoint from all other subtypes of T-Anything.
T-Lisp	The type of all non-Rhet lisp objects. It is disjoint from all other subtypes of T-Anything.
T-Atom	The type of Lisp Atoms. It is a subtype of T-Lisp.
T-List	The type of Lisp Lists. It is a subtype of T-Lisp.
T-Number	The type of Lisp Numbers. It is a subtype of T-Lisp.
T-Float	The type of Lisp Floating-point Numbers. It is a subtype of T-Number.
T-Integer	The type of Lisp Integers. It is a subtype of T-Number.
T-Rational	The type of Lisp Rationals. It is a subtype of T-Number.
T-Nil	The type of nothing. It is disjoint from all other types.

Table 2.3: Rhet's Predefined Types

For the below defined functions, the following note is applicable: If the keyword :**Candidate-Primary** is supplied and non-nil, the term will possibly be used as the primary member of it's canonical class (see equality section 5.8). Terms with no arguments are automatically candidate primaries and this cannot be suppressed.

Itype <*Typename*> &Rest <*Individual*> &Key *Candidate-Primary*
: which asserts that the individual(s) are of the indicated immediate type, *e.g.*, (**Itype 'Cat [A]**) asserts that the constant [A] is of type CAT, and is not in any known subset. See above for use of Candidate-Primary keyword.

Utype <*Typename*> &Rest <*Individual*> &Key *Candidate-Primary*
: This asserts that the individual is of the indicated type, *e.g.*, (UTYPE 'CAT [A]) asserts that the constant [A] is of type CAT, but may be a subtype, *e.g.* KITTEN. See above for use of Candidate-Primary keyword.

Dtype <*Individual*> &Rest <*Typename*> &Key *Candidate-Primary*
: This is similar to the **Utype** assertion, above, but also indicates that the individual cannot possibly be equal to any other Dtyped individual *of the type typename*. See also [Allen and Miller, 1990]. See above for use of Candidate-Primary keyword.

2.4 Structures in Rhet

The Rhet system supports reasoning about structured types (see [Allen and Miller, 1990] for an introduction). The following naming conventions[5], are used to distinguish the different kinds of objects and are observed by the Rhet system.

- T- ...— a type name
- R- ...— a rolename
- F- ...— the function named by a rolename
- C- ...— a constructor function

To define a subtype with roles, there are two options, depending on whether the objects of the new type are fully determined by the set of roles defined. Both of these enforce the restriction that the new type must be a subtype of an existing type. A type *T-U is predefined as the root of the Rhet type hierarchy. See section 6.3.2 for a description of how to retrieve role information about objects. See section 6.3.1 for a description of how to define roles on the type hierarchy.

[5] Actually, these have become stronger than mere conventions; the system parses the F- or C- off a function name in order to see if it's possibly a constructor function or role accessor. The type subsystem now tries to enforce the R- and T- conventions as well for structured type objects.

Chapter 3

Reasoning Modes

The unifier in Rhet has been augmented to allow two types of special unification dealing with equality and restricted variables.

3.1 Equality

The unification algorithm of Rhet has been modified so that when terms do not unify they can be matched by proving that the terms are equal. Any variables in the terms matched will be bound as needed to establish the equality. Equality statements are added to the system either by calling the Lisp function or by using the builtin Add-EQ[1]. For example:

(3.1) `(Add-EQ [president USA] [George-Bush])`

expresses a fact that is well known to most Americans. The axiom

(3.2) `(Add-EQ [add-zero 1] [1])`

expresses an infinite class of equalities. For example, `[add-zero [add-zero 1]]` equals `[1]`, as does `[add-zero [add-zero [add-zero 1]]]`, and so on.

The system provides, in an efficient manner, complete reasoning about fully grounded terms (*i.e.*, terms that contain no variables). The system will allow variables in queries (which may be bound to establish equalities).

The information derived from the Add-EQ axioms that are asserted is stored on a pre-computed table which is updated as Add-EQ axioms are added. Add-EQ axioms may not be deleted, however, since Add-EQ axioms may be added relative to a context, it is possible to pop the entire context.

[1]Add-EQ may not appear in the LHS of an axiom, thus its presence as a separate Lisp function rather than as something Assert-Axioms might add.

3.2 The Post-Constraint Mechanism

Rhet allows the user to specify that the proof of an form be delayed until the terms in it are completely bound. The user does this by enclosing the form within the function [Post], as in the axiom:

(3.3) [[F ?x] < [POST [MEMBER ?x (a very long list)]] [G ?x]]

POST takes an form as an argument. If the form is grounded then the proof proceeds as usual. Otherwise the variables in the form are bound to a function which restricts its value and the proof proceeds as though the proof of the form succeeded.

Restrictions on variables are implemented by setting a constraint on the variable that is presented as:

(3.4) [Any ?newvar [constraint ?newvar]].

Thus, give the above axiom, if we queried [F ?s], the [Post] mechanism would constrain ?s to

(3.5) [Any ?s0001 [MEMBER ?s0001 (a very long list)]].

This use of the special form Any is similar to the omega form used in Kornfeld [Kornfeld, 1983].

The Rhet unifier has been modified so that it knows about constrained variables. A term printed with the form [Any ?x [R ?x]] will unify with any term that satisfies the constraint [R ?x]. Again using the above axiom: after the POST succeeds, the proof continues with the subgoal

(3.6) [G [Any ?s0001 [MEMBER ?s0001 (a very ...)]]].

Now suppose that [G e] is true. Then we can unify these two literals if we can prove

(3.7) [MEMBER e (a very long list)].

Note that the constraint will be queried only once its variable is bound. Thus if [G ?c] were true above, the unification would succeed and

(3.8) [F [Any ?s0001 [MEMBER ?s0001 (a very long list)]]]

would be returned as the result of the proof. If [G [fn ?c]] were true instead, a recursive proof testing whether [MEMBER [fn ?c] (a very long list)] would be done and, if successful, the final result of the proof would be

(3.9) [F [fn [Any ?z [MEMBER [fn ?z] (a very long list)]]]].

NB: Certain builtins automatically post constraints on variables as needed. For example, if Rhet tries to prove [Distinct ?x B], it does not bind ?x to something that happens to not be [EQ?] to [B], but rather constrains ?x by changing it to: [Any ?x [Distinct ?x B]], which allows us to check for distinctness whenever ?x is actually bound. This avoids our having to pick a particular binding for ?x at the current time, which would likely force us to backtrack later should our guess be wrong.

3.3 Backward Chaining

This has adequately been described in the tutorial.

3.3.1 Defining Backward Production Axioms

Backward Chaining Axioms are added to the system via the **Assert-Axioms** Lisp function. As an example,

(3.10) (Assert-Axioms [[e ?f] <j])

would allow us to prove [e foo], while

(3.11) (Assert-Axioms [[w ?d] <j [r ?d]])

(3.12) (Assert-Axioms [[r d] <j])

would allow us to prove [w d]. For convenience, Assert-Axioms may take several Rhet axioms. Thus the above two assertions could have been made by

(3.13) (Assert-Axioms [[w ?d] <j [r ?d]] [[r d] <j]).

3.4 Forward Chaining

The prover has a forward production system in which the addition of new axioms adds new facts that are implied by the existing axioms. The general form of a forward axiom is as follows:

[[conclusion] <index <[condition]>* :forward <[trigger]>*]

After a Rhet axiom is added to the database it is checked to see if it matches any trigger pattern. A trigger must be form, but cannot be a Lisp predicate. If it matches, then using the binding list of the match the system tries to show that the conditions associated with the trigger are in the database. Note that the system does not try to prove the conditions (unless specified), but simply checks that they are in the database. If all the conditions can be shown to be in the database then the conclusion is added to the Rhet axiom list using the bindings collected in the process. Lisp predicates can be used in the conditions and in the conclusion, where they are called as in the backwards chaining system. The value returned by a Lisp predicate in the conclusion is ignored[2]. In adding a conclusion another trigger may be fired. To prevent infinite looping the forward chaining system will not add axioms that are already in the database. Rhet will do a simple consistency check before adding a forward chained assertion, namely, that it's inverse is not already asserted. This will not necessarily prevent contradictions from being discovered later, since the deductive closure is not kept.

[2]This would only be of use if the predicate had side-effects

3.4.1 Defining Forward Production Axioms

Forward Chaining axioms are added just like normal backward chaining axioms are, *i.e.* via **Assert-Axioms**. For example,

(3.14) `(Assert-Axioms [[w ?d] <r [r ?d] :forward [r ?d]])`

(3.15) `(Assert-Axioms [[r d] <s])`

will result in the derived fact

(3.16) `[[w d] <r]`

being added to the database, with the above statements for *support q.v.*. Using the atom :All for the trigger, or omitting the trigger list, adds a separate forward-chaining axiom for each of the forms in the condition list with that condition as the trigger. Thus each of the conditions is a trigger, *e.g.*,

(3.17) `(Assert-Axioms [[eq ?y ?z] <1 [eq ?y ?x] [eq ?x ?z] :forward :all])`

adds the following forward chaining axioms to the system:

(3.18) `[[eq ?y ?z] <1 [eq ?y ?x] [eq ?x ?z] :forward [eq ?y ?x]]`

(3.19) `[[eq ?y ?z] <1 [eq ?y ?x] [eq ?x ?z] :forward [eq ?x ?z]]`

Given these, the following additions:

(3.20) `(Assert-Axioms [[eq w e] <1])`

and

(3.21) `(Assert-Axioms [[eq r w] <1])`

causes the axiom

(3.22) `[[eq r e] <1]`

to be added to the system.

Naturally, a Lispfn may occupy the position of the predicate name in any of the conditions. The Lispfn succeeds if it returns a non Nil value.

The position of the predicate name may also be occupied by the atom :Prove whose argument is a form. This allows any of the conditions to call the Reasoner to prove the condition. (Note that normally conditions are not proved but just shown to be in the data base). The condition is true if the form can proven by the Reasoner. Any variables bound in the proof will be passed on to the next condition.

3.4.2 Truth Maintenance

The system also performs truth maintenance; *i.e.*, facts entered into the data base due to a forward-chaining axiom are removed when any supporting fact from the RHS is removed, if it has no other support. An example or two should clarify this: Given the axioms Mod

(3.23) `[[P ?x] < [Q ?x] [R ?x] :Forward]`

(3.24) `[[P ?x] < [Q ?x] [S ?x] :Forward]`

assume we make the following assertions:

(3.25) `[[Q A]]`

(3.26) `[[Q B]]`

(3.27) `[[R A]]`

which forward-chains `[[P A] <]`.

(3.28) `[[S B]]`

which forward-chains `[[P B] <]`.

(3.29) `[[R B]]`

which has no FC effect.

If we now retract any one of 3.23, 3.25, or 3.27, `[[P A] <]` would be retracted because it would have lost it's *support*. On the other hand, our conclusion of `[[P B] <]` is more robust, since it has alternate support. The only single retraction that would cause TMS to retract this conclusion would be 3.26.

Chapter 4

Built-In Predicates

This chapter documents the built-in predicates (referred to as Builtins) that are already defined in Rhet. Many also are defined as Lisp functions that can be called directly. For instance **Assert-Axioms**, the primary way of adding axioms to the system, is both a Lisp function (as defined above) and a Builtin. Some of the following builtins are described as being assertable. This means that it is legal for them to appear inside an `[Assert-Fact]`, `[Assert-Axioms]`, or on the LHS of a Forward Chaining axiom. The Builtins that follow are separated into three categories: those that deal with the type of an object or instantiating an object of a certain type; those that deal with properties of rhet terms, and those that deal with the proof process itself. New

4.1 Dealing with Instances

[**Add-EQ** Term1 Term2]

Both terms must be fully grounded. This predicate will add the equality specified. Note that the context the equality is added in may be explicitly specified by **Assert-Axioms**, or the modal operators if this is in a horn clause. Since equalities cannot be retracted, the `[Assume]` builtin can be used in conjunction with it. Note that this function is also available directly from Lisp.

[**Add-InEQ** Term1 Term2]

Both terms must be fully grounded. This predicate will add the inequality specified. Note that the context the inequality is added in may be explicitly specified by **Assert-Axioms**, or the modal operators if this is on the RHS of a horn clause. Like equalities added via the `[Add-EQ]` function or builtin, inequalities cannot be retracted. Note that this function is also available directly from Lisp.

[**Add-Role** Instance *&Rest* <Role-Name Value>*]

Adds that object Instance has role Role-Name with value Value. All arguments must be fully grounded. Note that Role-Name is expected to be a Lisp atom, *e.g.* :**R-Name**.

[**Assert-Relations** Instance Relation-Keyword *&Optional* (Accessor-String (String Relation-Keyword))]

Mod

This builtin presumes a specific usage for relations: that of constructor functions for other types. For each of the relations defined on Relation-Keyword for the type of the Instance, the relation is generated and set (via **Add-Eq**) to an accessor. This accessor is by default the term `[relation-# ?self]`.

An example should clarify this:

```
(4.1)    (Define-Subtype 'T-HACK-1 'T-U
                 :Roles '((R-ACTOR T-ANIMATE))
                 :Initializations '([Assert-Relations ?self :steps])
                 :Relations '((:STEPS [C-EAT-PIZZA [F-ACTOR ?self]]
                                     [C-DRINK-BEER [F-ACTOR ?self]]
                                     [C-HACK-COMPUTER
                                        [F-ACTOR ?self]])))
```

that is, given the above 3 STEPS being defined on type T-Hack-1, (locally, in this case, but they might have been inherited), when we construct an instance of T-Hack-1, say `[HACK1]`, then our initializations are run, and `[Assert-Relations HACK1 :steps]` will succeed and generate three accessors, `[Steps-1 HACK1]`, `[Steps-2 HACK1]`, and `[Steps-3 HACK1]`, each `[EQ?]` to whatever was on the STEPS relation in those positions, with ?Self appropriately bound. In this case, we would have asserted:

```
(4.2)    (Add-Eq [Steps-1 HACK1] [C-EAT-PIZZA [F-ACTOR HACK1]])
```

```
(4.3)    (Add-Eq [Steps-2 HACK1] [C-DRINK-BEER [F-ACTOR HACK1]])
```

and

```
(4.4)    (Add-Eq [Steps-3 HACK1] [C-HACK-COMPUTER [F-ACTOR HACK1]])
```

In general, if this relations form is a constructor function, as it is in this example, the construction is done at this time as well. A more complete example of possible use is given in section 6.21. A special usage is when Accessor-String is passed as :T. In this case, the relation defined should itself be an ALIST, whose key is an atom whose printname will be used as the accessor-string, and whose value will be the interesting constructor function. To make this clear, let me restate the above **Define-Subtype** as follows:

```
(4.5)    (Define-Subtype 'T-HACK-2 'T-U
                 :Roles '((R-ACTOR T-ANIMATE))
                 :Relations '((:STEPS (STEP-1 . [C-EAT-PIZZA
                                                  [F-ACTOR ?self]])
                                     (STEP-3 . [C-DRINK-BEER
                                                  [F-ACTOR ?self]])
                                     (STEP-5 . [C-HACK-COMPUTER
                                                  [F-ACTOR ?self]]))))
```

Note that this is similar to our first example, with the exception that our accessors would now be [STEP-1 HACK1], [STEP-3 HACK1], and [STEP-5 HACK1]. The advantage of the latter usage is mainly so the steps can be sparsely named; subtypes could then supply other steps in the sequence, or specialize steps created by the parent type. *I.e.*:

(4.6)
```
(Define-Subtype 'T-HACK-LISPM 'T-HACK-2
                :Relations '((:STEPS (STEP-1 . [C-EAT-GOURMET-PIZZA
                                                 [F-ACTOR ?self]])
                                     (STEP-2 . [C-CHEW-HUNAN-PEPPER
                                                 [F-ACTOR ?self]])
                                     (STEP-3 . [C-DRINK-MOLSON-BEER
                                                 [F-ACTOR ?self]])
                                     (STEP-4 . [C-READ-NETNEWS
                                                 [F-ACTOR ?self]])
                                     (STEP-5 . [C-HACK-LISPM
                                                 [F-ACTOR ?self]]))))
```

Remember that [Assert-Relations] supports a particular kind of usage of the relations field; other usages are possible. If the Instance does not have the specified relation defined, [Assert-Relations] succeeds.

[Contradiction Instance]

Signal that the Instance is contradictory. This builtin is assertable. Normally, it would be used on the LHS of an FC axiom. An user might want to signal an instance as contradictory if they are writing a reasoner in Rhet, and one of the user's consistency tests fails. For instance:

(4.7)
```
[[Contradiction ?x] ¡ [not [Lispfn-predicate ?x]]
                     :forward [Holds ?x]]
```

The **Contradiction** builtin simply makes accessible to the user the same signalling mechanism that non-foldable constraints use. See also [Miller, 1990a] for a more detailed example of signalling and using contradictions in Rhet.

[Dtype <Type Descriptor> *&Rest* Form*]

Sets the Form(s) (any Rhet object) to be of the type described, as for **Utype**. Note that this type is NOT defined to be the immediate type. It is legal to later change the type (via **Utype** or **Itype**) to a more specific type. It signals an error to set the type of an atom to be something that is not a subtype of it's existing type, or to be anything other than it's immediate type should that have been declared (see **Itype**). See also **Utype**. The difference between **Dtype** and **Utype** is that the former also declares the Form to be unequal (see **Add-InEQ**) to any other Form that is **Dtyped** for this specific type. This builtin is assertable.

[EQ? Term1 Term2]

Succeeds if Term1 equals Term2 (*i.e.*, they unify). Posts constraints as needed.

[Expand-Constructor Constructor-Function]

New

Normally, constructors are only expanded when the constructor term is involved in an equality. To expand a constructor is to assert the equalities between role accessors and the actual slots on the function. This builtin forces immediate expansion.

[Itype <Type Descriptor> *&Rest* <Term>]

Sets the terms (any Rhet object) to be of the immediate type described. Note that this type is immutable. That is, given a type hierarchy that splits the type bird into flying-birds and non-flying birds, to have asserted an object to have the immediate type of bird makes it illegal to further classify it into flying or non-flying via the type system. It is not legal to later change the type (via **Utype** or **Itype**) to a more specific type. It signals an error to set the type of an atom to be something that is not a subtype of it's existing type (see **Utype**), or to be anything other than it's immediate type should that have been declared. See also **Utype**, **Dtype**. This builtin is assertable.

[NotEQ? Term1 Term2]

This will succeed if Term1 and Term2 are provably unequal. That is, either their types would not unify, they are distinguished elements of the same type, or they have been asserted to be **Add-InEQ**. Otherwise it fails.

[Relation-Form? Thing Relation Type]

This builtin succeeds if Thing is Equal to an element of the Relation list on type Type, which should be a structured type. If Thing is an unbound variable, this builtin will bind it to (a copy of) successive elements of the Relation-List for Relation on Type. It doesn't currently handle unbound variables passed for Relation or Type.

[Relation-List List Relation Type]

Succeeds if List is Equal to the Relation list on Type, a structured type. If List is an unbound variable, then it binds it to (a copy of) the Relation-List for Relation on Type. Again, it doesn't currently handle unbound variables passed for Relation or Type.

[Role Instance Role-Name Value]

Succeeds if the Rhet object Instance has role Role-Name with value Value. Value is a Rhet object, Role-Name is a Lisp Atom. See also **Add-Role**.

[Role? <Role-Name-Atom or Variable> <Type-Descriptor or Variable>]

Succeeds if the <Role-Name-Atom> is a Role defined or inherited by the <Type-Descriptor>. The Type Descriptor should be either an atom or a list, just like what would normally appear after the "*" on a typed variable (except for the colon needed to distinguish the term as an atom rather than a function term). For example, `:T-Lisp` would be used for type T-Lisp, and `(T-A T-B)` would be used for the intersection of type T-A and T-B. If the Role Name Atom position is an unbound Variable, the function will successively bind it to the roles of the Type-Descriptor. If the Type-Descriptor position is an unbound variable, it will be successively bound to all declared types with the appropriate role defined. If they are both variables, they will map over all roles on all types with roles. Some examples:

```
(4.8)   (define-subtype 't-transfer-object 't-u
                       :roles '((r-agent t-animate)
                                (r-benefactor t-animate)
                                (r-object t-thingy)
                                (r-time t-time)))
```

```
Rhet-> (prove-all [Role? ?x*T-Lisp :t-transfer-object])

([ROLE? :R-TIME :T-TRANSFER-OBJECT]
 [ROLE? :R-OBJECT :T-TRANSFER-OBJECT]
 [ROLE? :R-BENEFACTOR :T-TRANSFER-OBJECT]
 [ROLE? :R-AGENT :T-TRANSFER-OBJECT])

Rhet-> (prove-all [Role? :R-BENEFACTOR ?x*T-Lisp])
([ROLE? :R-BENEFACTOR :T-TRANSFER-OBJECT])
Rhet->
```

[**Skolemize** <Variable> <Type-Descriptor>]

Binds Variable to a newly instantiated term of type Type, asserting any constraints on the Variable as needed. New

[**Type-Relation** Type1 Relation Type2]

Succeeds if Type1 has relation Relation to Type2. The possible relations are: :Subtype :Supertype :Equal :Compatible[1] :Exclusive and :Nondisjoint. See table 6.1 for more details. New

[**Utype** <Type Descriptor> *&Rest* <Form>]

Sets the Form(s) (any Rhet object) to be of the type described. Note that this type is NOT defined to be the immediate type. It is legal to later change the type (via **Utype** or **Itype**) to a more specific type. It signals an error to set the type of an atom to be something that is not a subtype of it's existing type, or to be anything other than it's immediate type should that have been declared (see **Itype**). See also **Dtype**. This builtin is assertable.

4.2 Dealing with Other Terms

[**Assert-Axioms** <Axiom1> ...<AxiomN> *&key* Context Justification]

Adds the specified axioms to the data base for the specified predicate (see sections 3.3.1 and 3.4.1). It returns non-nil, thus succeeds when found on the RHS of a horn clause. All logic variables in the new axioms that appear outside the axiom assertion in the current environment will be replaced by their values before the new axioms are

[1] (OR :Subtype :Equal)

added. If the context parameter is specified, the string passed will indicate to the system which context facts should be asserted relative to. It will otherwise default to the default context, which the user can set programmatically. Note that this is one of the rare Rhet predicates that is available directly from Lisp, as Assert-Axioms. Note that in general the facts passed to `[Assert-Axioms]` may not be builtins, however, certain builtins and lispfns that have been declared to be assertable (see 5.7.3 for more information on assertable lispfns) may appear. For example, `[And]` may appear here, though it isn't really necessary; `[RFormat]` may appear, again, having the same functionality as if it appeared without being wrapped by the `[Assert-Fact]`, and modal operators may appear. Assertable builtins are documented as such.

[**Assert-Fact** <Fact1> ... <FactN> *&key* Context Justification]

Adds the specified facts to the data base for the specified predicate. It returns non-nil, thus succeeds when found on the RHS of a horn clause. All logic variables in the new facts that are bound in the current environment will be replaced by their values before the new facts are added; if any unbound variables are left after this process, `[Assert-Fact]` will signal an error. If the context parameter is specified, the string passed will indicate to the system which context facts should be asserted relative to. It will otherwise default to the default context, which the user can set programmatically. A runtime error is generated should any variables remain unbound. Note that in general the facts passed to **Assert-Fact** may not be builtins, however, certain builtins and lispfns that have been declared to be assertable (see 5.7.3 for more information on assertable lispfns) may appear. For example, `[And]` may appear here, though it isn't really necessary; `[RFormat]` may appear, again, having the same functionality as if it appeared without being wrapped by the **Assert-Fact**, and modal operators may appear. Assertable builtins are documented as such.

[**Bound** Term]

Succeeds only if Term is a bound variable. It fails on any other term. For example, `[bound ?x]`, where ?x is bound to the Fact FOO succeeds, while `[bound boy]` or `[bound ?x]` fails. Distinguish this from `[NOT [VAR ?x]]`.

[**Distinct** Term1 Term2]

Succeeds if both terms are fully grounded, but to different atoms. If a term is not fully grounded, this posts a constraint on the variable(s) and succeeds.

[**Fix-Cardinality** Set]

New

The cardinality of a set is fixed to be the current known contents. Has no effect on sets that already have a known cardinality.

[Ground Term1]

Succeeds if Term is a fully grounded term, *i.e.*, it contains no unbound variables. Mod

[Identical Term1 Term2]

Succeeds if Term1 and Term2 are structurally identical, *i.e.*, if they unify without assignment of variables or the equality mechanism. For example, `[IDENTICAL A A]` succeeds, and `[IDENTICAL A ?x]` or `[IDENTICAL ?x ?y]` (both unbound) fails.

[Lisp-EQ? Term1 Term2]

Succeeds only if Term1 and Term2 are EQ in the Lisp sense, that is, they denote the same internal object. New

[Retract Form]

Retracts all BC axioms whose head unifies with Form. All facts that unify are also removed. This builtin is also available from lisp.

[Subtype? Subtype Supertype]

Succeeds if the subtype is indeed a subtype of the supertype. If the form is not ground, it is set to successive types that satisfy the relationship. New

[Type? Form <Type Descriptor or Variable>]

Succeeds if either the Form is of the described type, or if the form is a variable, if it can be set to the type. If the type parameter is a variable, it is set to the "best" available type we have for the object. Normally this will be the immediate type, if it exists, otherwise, the most immediate type declared for atomic objects, or best fit for predicates. See also Utype. The Type Descriptor should be either an atom or a list, just like what would normally appear after the "*" on a typed variable (except for the colon needed to distinguish the term as an atom rather than a function term). For example, :T-Lisp would be used for type T-Lisp, and (T-A T-B) would be used for the intersection of type T-A and T-B. One can also use the immediate type representation, *e.g.* #!T-Lisp and #!(T-A T-B) respectively. New

[Unify Term1 Term2]

Succeeds if the two terms unify without equality. This is faster than `[EQ?]`. It can also be used to determine when two terms are equal but distinct, as in

(4.9) `[And [EQ? ?x ?y] [Not [Unify ?x ?y]]].`

[Var Term]

Succeeds only if Term is an unbound variable.

4.3 Dealing with Proofs

[And Form*]

Succeeds only if all forms can be proven. `[AND]` succeeds. If `[Cut]` is encountered as a Form, backtracking past it cuts out of the entire AND, but not necessarily out of the entire rule. A singular AND on the RHS of a horn clause has the same semantics as a normal horn clause's RHS. Note that [And] will evaluate it's arguments specifically in the order supplied, and will short-circuit evaluation as needed. Thus the clause [And [P ?x] [Q ?x]] will not cause evaluation of [Q ?x] if [P ?x] cannot be proven. This builtin is assertable; all of the Forms in the argument list must themselves be assertable.

[And* <Form>*]

A macro form for [AND [CUT] Form1 [CUT] Form2 ... [CUT] FormN]. That is, any backtracking needed at all inside of the AND* causes it to fail. This builtin is assertable; it acts just like [And] does in an assertion.

[Bagof Var1 Form Var2]

This is like [Setall], but it allows duplicates in Var1. This will typically be faster than [Setall]. It succeeds if Var1 is set to the list of all non-unique assignments to var2 that satisfy Form[2]. For example, [Bagof ?x [P ?y ?z] ?z] would set ?x to a list of all things that have been asserted as arguments to predicate P that make it true. For example, given [P A B], [P B C], and [P D C] ?x would be bound to ([B] [C] [C]).

[Call Lisp-Expression]

Calls the interpreter on the Lisp expression passed. This allows the user to call arbitrary Lisp expressions without having to use Declare-Lispfn on them. Note that the argument must be a Lisp EXPRESSION not a function[3]! (*E.g.* (FOO X) is OK, #'FOO isn't). In other words, pass a Lisp list. Unbound variables will remain as variable structures, so the user must either handle these via [Setvalue] and [Genvalue], or make sure the variables are bound (using, *e.g.*, Bound) before allowing a Call to be processed.

[Cond Condbody]

What you would expect if you were writing in Lisp. The predicate, however, must be a Rhet predicate (or an appropriately declared Lisp predicate) and the first provable predicate form encountered causes the associated action forms to be proved. If any action fails, Cond immediately fails, but if a predicate fails, Cond tries the next predicate in its list. Cond succeeds if some predicate succeeds, and no associated executed action fails. If no successful predicate is found, Cond fails. Note that for backtracking purposes,

```
(4.10)   [COND ([pred1]
               [actionset1-1]
               [actionset1-2] ...)
```

[2] By this we mean that the variable Var2 is assigned to a ground term, such that the Form with Var2 so bound is found in the KB directly, as opposed to via backward chaining.

[3] This is, in fact true, of all Rhet functions documented as taking a Lisp expression as an argument. This allows the Rhet system to bind variables in the arguments of the function appropriately, which it could not do if it were handed a compiled object. If the user is concerned with runtime efficiency, or wants to be able to backtrack, they should get a copy of the Programmer's Manual and write a Builtin instead.

```
([pred2]
 [actionset2-1] ...)
...]
```

acts as follows:

```
(4.11)    [OR [AND [pred1]
                   [CUT]
                   [actionset1-1]
                   [actionset1-2] ...]
              [CUT]
              [AND [pred2]
                   [CUT]
                   [actionset2-1] ...]
              [CUT]
              ...]
```

The idea is that once a pred has succeeded, the axiom can be considered to consist only of the actions associated with that pred. No other pred will be tested on backtracking, though actions can be backtracked through[4]. In PROLOG, separate clauses with a cut after their first form (a typical idiom) also gives this behavior.

Note that the [Win] and [Fail] builtins can be used to good effect inside of a Cond.

[Cut]

The Cut symbol. It has no effect until Rhet tries to backtrack past it, and then the prover immediately fails on the subproblem it was working on. An alternate definition: cut always succeeds, and when executed, removes all choice points in the proof from the point at which the predicate which appears in the head of the axiom containing the cut was selected to the current point of the proof.

[Fail]

This predicate is always false.

[Forall <Variable / list> Form1 FormN*]

[5] Succeeds if for all variables in <Variable / list> (which can be an atomic variable) as constrained by Form1, each form of FormN* succeeds. For example, [Forall ?y [P ?y] [Z ?y]] says that for all assignments to ?y (*e.g.* if [P A] is in the KB, then A) Z of it must be true for the form to succeed.

In general these semantics are a little flaky, because [Forall (?y ?z) [P ?y] [Z ?z]] is closer to

[4]Thus, it would not be a good idea to use global variables in the predicate, since only one binding will be made, even in a Prove-All.

[5]Current Status: Known to work only if no form contains a variable not in the variable list, and all variables are used in Form1.

$$\forall y\ P(y) \Rightarrow \exists z\ Z(z) \tag{4.1}$$

when we probably meant

$$\forall y, z\ P(y) \Rightarrow Z(z). \tag{4.2}$$

We are currently looking at expanding Forall to have the more intuitive definition, and adding an Exists builtin so such relationships can be made explicit.

[Forall! <Variable / list> Form1 FormN*]

New

Just like Forall, except when it appears as a constraint or initialization. Forall is by default non-foldable and non-monotonic, while Forall! is both foldable and Monotonic. This is useful if, for instance, the body of the Forall is used to initialize slots on a type instance. For example,

```
(4.12)  (DECLARE-SET-TYPE 'T-HUMANS 'T-ORTHODOX-SET 'T-HUMAN)

(4.13)  (DEFINE-SUBTYPE
            'T-PARENT 'T-HUMAN
            :ROLES '((R-CHILDREN T-HUMANS))
            :CONSTRAINTS '([FORALL! ?X
                                   [MEMBER ?X [F-CHILDREN ?SELF]]
                                  [COND ([TYPE? ?SELF #!T-MALE]
                                         [EQ? ?SELF [F-FATHER ?X]])
                                        ([WIN]
                                         [EQ? ?SELF [F-MOTHER ?X]])]]))
```

might be used to set the father or mother slot of each of the children for the instance of a parent.

[Genvalue <Variable / list> Lisp-Expression]

Sets the Rhet variable <Variable> to first value in list returned by evaluating the Lisp-expression. Other values are used for backtracking. If the Lisp expression returns more than one value (in the Common Lisp sense) then the Car of each of these values will be assigned to each variable in the list, with extras used for backtracking (*i.e.* successive Cadr). Extra variables are set to Nil. Extra values returned are ignored.

[Member Term1 List]

Succeeds if Term1 unifies with a term on the List.

[Not Form]

On the LHS of an axiom it indicates the axiom is used to disprove the enclosed horn clause (or unenclosed predicate, *i.e.* `[NOT [P X]]` vs. `[NOT P X]`). On the RHS of an axiom, it indicates that the enclosed horn clause or unenclosed singular term is to be disproved using whatever proof mode is in force (*i.e.* complete if prove complete was engaged, or simple if simple or default reasoning modes are in force).

[Or *&Rest* Form*]

Succeeds if any of the Form* can be proved. `[OR]` fails. `[Cut]` is not caught by this form, unlike `[And]`, so backtracking past a CUT in this form causes the rule in which it is used to fail. Note that Or will evaluate it's arguments specifically in the order supplied, and will short-circuit evaluation as needed. Thus the clause `[Or [P ?x] [Q ?x]]` will not cause evaluation of `[Q ?x]` if `[P ?x]` is true.

[Post Form]

Succeeds if the Form is ground and can be proven, fails if the Form is ground and cannot be proven, and otherwise constrains all variables in the form such that they can only be unified with objects that would have allowed the form to succeed. See section 3.2 on the Post-Constraint Mechanism in the tutorial for a further description. Note that several other Rhet builtins use an implicit POST for certain arguments. See, *e.g.* DISTINCT.

[Prove Form]

Succeeds if the Form can be proven using backward chaining. Normally this would only be used within an FC axiom, since BC axioms would normally use these semantics for each form on their RHS.

[Setall Var1 Form Var2]

Succeeds if Var1 is set to the list of all unique assignments of Var2 that satisfy Form[6]. For example, `[Setall ?x [P ?y ?z] ?z]` would set ?x to a list of all things that have been asserted as arguments to predicate P that make it true. For example, given `[P A B]`, `[P B C]`, and `[P D C]` ?x would be bound to ([B] [C]). It is similar to `[Bagof]`, which is more efficient.

[Setvalue <variable / list> <Lisp-Expression>]

Sets the Rhet variable <variable> to the value of the Lisp expression <Lisp-Expression>. Any logic variables in <Lisp-expression> are replaced by their logic bindings before Lisp evaluation. If the Lisp expression returns more than one value (in the Common Lisp sense), then the values are assigned to each variable in the list in turn. Extra variables in the list are bound to Nil. Too few variables cause the extra values to be discarded.

[Set-Setvalue <variable / list> <Lisp-Expression>]

Sets the Rhet variable <variable> to the value of the Lisp expression <Lisp-Expression> as a Set, rather than a list. Any logic variables in <Lisp-expression> are replaced by their logic bindings before Lisp evaluation. If the Lisp expression returns more than one value (in the Common Lisp sense), then the values are assigned to each variable in the list in turn. Extra variables in the list are bound to Nil. Too few variables cause the extra values to be discarded. For example: `[Set-Setvalue ?x*t-set (quote (a b c))]` succeeds binding ?x to {`a b c`}. New

[6] By this we mean that the variable Var2 is assigned to a ground term, such that the Form with Var2 so bound is found in the KB directly, as opposed to via backward chaining.

[Unless Form*]
Succeeds if any Form in the forms given cannot be proven, or can be proved as [NOT Form]. This gives us proof by failure. Note that variables change in interpretation in the Unless function, *e.g.*, if we are given the fact that [P A] is true, then

[Unless [P B]] will succeed,

[Unless [P A]] will fail as expected, but

[Unless [P ?x]] also fails, since [P ?x] can be proven.

[Win]
Always succeeds. This is mainly needed as a predicate inside of [Cond] forms.

4.4 I/O

[RFormat Stream Control-String *&Rest* Form*]
The values of the list Form* are printed according to the Control-String on Stream. This is just like the Common Lisp Format function. This builtin is assertable. Normally Stream would be :T for the default, but can also be a list that will evaluate to a stream. That is, this will translate into a call to (Apply #'Format (List* (Eval Stream) Control-String Form*)).

[Rprint Term1 ... TermN]
The values of Term1 through TermN are pretty-printed on successive lines. This builtin is assertable.

[Rterpri]
Prints a line feed and even a carriage return. This builtin is assertable.

4.5 Numbers

Some specific builtins are included for dealing with objects that are of, or can be coerced to, the *T-Number type, which include the types *T-Integer, *T-Float, and *T-Rational, which capture their respective common lisp types.

[:= Variable Expression]
The Variable, which must be of type *T-Number, is bound to the value of the Expression after evaluation. The Expression would normally be a list, using LISP operators on Rhet variables and numeric constants to produce a result. This builtin succeeds if the unification succeeds.

[== Expression1 Expression2]
Succeeds if the numeric value resulting from evaluating Expression1 is EQL to the numeric value resulting from evaluating Expression2.

[=/= Expression1 Expression2]
As ==, above, but succeeds if the values are not equal.

[< Expression1 Expression2]
As ==, above, but succeeds if the value of Expression1 is less than Expression2 (using lisp's < operator).

[> Expression1 Expression2]
As ==, above, but succeeds if the value of Expression1 is greater than Expression2 (using lisp's > operator).

[<= Expression1 Expression2]
As ==, above, but succeeds if the value of Expression1 is less than or equal to Expression2 (using lisp's <= operator).

[>= Expression1 Expression2]
As ==, above, but succeeds if the value of Expression1 is greater than or equal to Expression2 (using lisp's >= operator).

4.6 Context Manipulation

In addition general modal operators for Belief are defined, as well as some builtins for modifying proof context at runtime:

[AB Form]
This asserts the Form, (if it appears in Assert-Axioms), or tests the Form (if it appears in the RHS of a horn clause) into the belief space A. A can be any alphabetic character, except M, It may also appear on the LHS of a horn clause, with the usual semantics. This has the semantics described in the appropriate section of the tutorial, [Allen and Miller, 1990]. These may be nested at will, for convenience, without parenthesis, much as Lisp Cadr, that is

(4.14) `[SBHBTBQB[P] <]`

would translate to an assertion that relative to S's beliefs about H's beliefs about T's beliefs about Q's beliefs, P holds. Note that it is possible to move to the context set up by a belief operator, via the Operator Lisp function, but it's best to use the modal operator, whenever possible. This builtin is assertable.

[Assume *&Rest* Form*]
Creates a subcontext to the default one and attempts to prove each of the Forms, which can be an arbitrary series of clauses, as on the RHS of a horn clause. Normally one or more of these clauses would be an assertion, as of equality. The ASSUME succeeds or fails according to the same rules as an axiom's RHS allowing the LHS to be considered proved. Note that ASSUME, like all predicates, can be wrapped in one or more modal operators, with the exception of an MB operator[7]. Note also that

New

[7] *I.e.* `[MB [ASSUME [foo] [bar]]]`, which would read "assuming that it is mutually believed that foo, can bar be proved?" While supporting such a beast would be nice, the user must handle such cases manually, somehow.

while ASSUME creates a context, it is impossible to move to this context, or use it as an argument to any function that takes a context as an argument[8].

[MB Form]

Like AB, but uses a special "Mutual Belief" space. To assert `[MB[P] <]` it is derivable that `[SB[P]]` and `[HB[P]]` as well as the full belief closure between these agents, (*e.g.* `[SBHBSBHBSB[P]]`). Restriction: the MB operator may not be used in an ASSUME clause. This builtin is assertable. MB is used for mutual belief amoung all agents. There are relative MB spaces as well, e.g. `[ABHBMB[P] <]` does not imply `[AB[P]]`, `[HB[P]]`, or even `[MB[P]]`, but does imply, for instance, that `[ABHB[P]]`.

[With UContext *&Rest* Form*]

Similar to ASSUME, except that instead of building an arbitrary UContext, it creates a named UContext (if it didn't exist) and sets the proof UContext to that UContext. It then attempts to prove each of the Forms, succeeding in the same fashion that AND succeeds. UContext must be a string, the name of a UContext.

Note that all such belief statements are relative to the current DEFAULT UContext.

[8] clear?

Chapter 5

Programmatic Interface

The main thing to remember is that when you build a system to use Rhet as your KR engine, you should have your system's package USE the packages Assert, Query, and UI (at least). That way, your system has access to Rhet's exported functions that are intended to be used and unlikely to change without using a package prefix. This can make maintenance easier, as between Rhet releases if something breaks, you may be able to quickly localize the problem to a call you make to some "internal" Rhet function that we didn't promise to keep constant (and hence didn't warn you would change)[1].

5.1 Manipulating Facts

5.1.1 Adding and Deleting Facts

Assert-Axioms *<Axiom1>* ... *<AxiomN>* &key *Context Justifications*

Adds the specified axioms to the data base. It returns a list of the axioms (or facts) so added, thus succeeds when found on the RHS of a horn clause. For axioms, that is, if there is a RHS, one would normally use **DefRhetPred**, however. If the context parameter is specified, the context passed will indicate to the system which context axioms should be asserted relative to. Normally one would use the **UContext** and or **Operator** functions to convert a string into a context for **Assert-Axioms**. It will otherwise default to the default context, which the user can set programmatically[2]. The Justifications are added to any facts added via this form. Justifications are created by appropriately[3] massaging the proof tree from a BC proof. See also the TMS section of the tutorial. Note that this function is also available as a builtin.

Retract *<Fact>* &Key *Context*

Retracts the (previously asserted) Fact passed in the context Context. It is permissible

[1]Not that we don't live to pull the rug out from under you with the documented interface, but at least you have some reason to complain then. ⌣

[2]It is the value of *Default-Context*.

[3]There should be a function, eventually, to do this, particularly since we haven't specified what is an appropriate munging.

to, for example, assert [Foo] in a context, and retract it in a child context. This would make [Foo] true in the asserted context, and all child contexts above the context it was retracted in. *I.e.* if we add context TC as a child of T, and TCC as a child of TC, and assert [Foo] in T and retract it in TCC, we would still see foo as true in contexts T and TC, but undefined in TCC[4]. Note that a version of this function is also available as a builtin.

Retractall <*Form*> &Key *Context*

Retracts all the facts in the database whose conclusions unify with the specified form. The predicate name must be specified in the form. If an atom is given as a pattern, it will be interpreted as a predicate name and all axioms for that predicate will be deleted. For example, (Retractall [P A ?x]) retracts all facts whose head unifies with [P A ?x] (*e.g.*, [P ?x ?z], [P ?x B], [P A B]), and (Retractall 'P) retracts all facts with head P[5]. Note that the builtin [Retract] can also be used with this effect.

Clear *RE Context*

Retracts all facts in the database with an index that could be generated by the regular expression. See 5.1.1.1, below, for more information on acceptable regular expressions.

5.1.1.1 Regular Expressions

Rhet allows indexes, for commands that accept an index as an argument, to be a regular expression. Using the operators[6], "∧"[7] for "and" "∨"[8] for "or", "∈∈"[9] for the empty string, "[" and "]" to delimit subexpressions, and "*" for Kleene closure in a string, once can specify a miniature "program" to Rhet for finding indexes taken as a string. So, for example, the index "$< \wedge [f \vee b] \wedge oo$" would match fact indices of either "<foo" or "<boo".

5.1.2 Accessing Facts

Find-Facts &Rest *Forms*

Returns all assertions of form [<conclusion>] or [<conclusion> <index>] that unify with Form. Thus to find all facts that assert that P is true (or false) of something, we could use (Find-Facts [P ?x]). If the data base contained the facts [P A], [P B <a], [[P D] <b [Q R]], then the query would return (([[P B] <a] [[P A]]))[10].

Find-Facts-By-Index *Index* &Key *Context*

Returns all facts in the context whose Indices match Index. Given the data base above, (find-facts-by-index '<) would return ([[P A] ¡]), while (find-facts-by-index "<a") would

[4] In fact, we can assert [NOT Foo] in TC, and have Foo true in T, false in TC, and unknown in TCC.

[5] This is equivalent to specifying a form of [P &Rest ?x].

[6] A not operator and "+" should be added.

[7] Symbol-w on the Symbolics or Explorer keyboards

[8] Symbol-q

[9] Symbol-Shift-e

[10] Find-Facts will *never* find axioms, only asserted true or false facts.

return ([[P B] ¡a]). Find-Facts-By-Index may be passed an Index either as an atom, or a string. An atom will search for exact matches to the atom, while a string may be any regular expression, used as described in section 5.1.1.1.

Find-Facts-With-Bindings &Rest *Forms*
Same as find-facts except that it returns the variable bindings as well in the format ((<axiom> <binding list>)*) for each form. For example, with the above three axioms for P, the query (find-facts-with-bindings [P ?x]) would return ((([[P B] <a] (?x [B])) ([[P A]] (?x [A])))).

Find-Fact-References &Rest *Forms*
Returns all facts in the context that reference (has as an argument) all of the supplied Forms.

5.2 Manipulating Axioms

5.2.1 Adding and Deleting Axioms

Note that unlike Facts, Axioms may not be deleted from a particular context that is different from the context they were asserted in.

Assert-Axioms *<Axiom1>* ... *<AxiomN>* &key *Context*
See 5.1.1

Clear-Axioms
Delete all axioms defined by the user. Facts remain intact.

DefRhetPred *Predicate-Name* (Argument-Lambda-List) &Body *Body*
Defines a Rhet predicate Predicate-Name that takes arguments according to the Argument-Lambda-List. Body is either a series of indices and RHS definitions, or lisp code. These may not be mixed.

More specifically, the Argument-Lambda-List may be made up of the following:

1. & keywords, specifically:

   &Any the following variables (until the next & keyword) may be bound or unbound when the predicate is invoked, or may be bound to terms that are not fully ground. This is the default when no & keyword appears.

   &Bound the following variables are guaranteed by the programmer to be bound when the predicate is invoked. It is an error to attempt to prove the predicate without passing a fully ground term in this position. If (Declare (Optimize Safety)) appears, erroneous usage will signal an error[11]. If forms follow this keyword, variables embedded in the forms must be bound.

[11] Future functionality

&Forward this must be the last &Keyword specified. If present, the **DefRhetPred** defines FC axiom(s) rather than BC. The form(s) following this key are the trigger(s). If none are present, it is the same as having specified :All as the trigger. No unbound variables may appear in the trigger. The other part of the lambda list becomes a pattern for the fact to be asserted if the body of the **DefRhetPred** is proved[12].

&Unbound the following variables are guaranteed by the programmer to be unbound when the predicate is invoked. If forms follow the keyword, variables within the forms must be unbound. It is an error to attempt to prove the predicate with any of the variables bound. If **(Declare (Optimize Safety))** appears, erroneous usage will signal an error[13].

&Rest the following variable must be of type T-List (it will be changed if it is not), and will be bound to all the remaining arguments of the predicate when attempting to prove it. This is just like the normal usage of &Rest in forms.

2. Rhet variables, which have the properties of the closest preceding & keyword, as above, or &Any by default.
3. Rhet forms, which are pattern matched against the form being proved to see if this predicate is applicable.

The body then consists of the following:

1. If the first object is a string, it is considered a documentation string for the predicate.
2. If the first object (after the optional documentation string, if any) is a **Declare** form, it is taken to be declarations about the predicate, as for CL. All CL declarations are legal for predicates, if the embedded predicate definitions is a lisp function. If they are Rhet RHS terms instead, most such declarations are legal, but ignored. Other declarations that are legal include Foldable. Non-Foldable, Monotonic, and Non-Monotonic. See the definition for **Define-Subtype**, 6.3.1, for more information on these declarations, and how they interact with the constraints of the structured type subsystem.
3. After these optional declarations, if the next object is an index (*i.e.* it begins with the character "<"), then the following objects must all be legal Rhet terms or indices. All terms until the next index or the end of the **DefRhetPred** will make up the RHS of an axiom (either BC or FC depending on whether the &Forward key was present in the Argument-Lambda-List), whose LHS is constructed from the Argument-Lambda-List.

   If another index is found, it begins the definition of a new BC/FC axiom, with the same LHS as defined by the Argument-Lambda-LIst.

[12]Currently, it is an error to have both &forward, and the body containing a lisp function.

[13]Future functionality

It is possible to have multiple **DefRhetPred** forms for the same predicate providing the Argument-Lambda-List can be distinguished between them, and all of them use this sort of body definition.

4. After the optional declarations, if the next object is not an index, the remainder to the body is taken to be an implicit **Progn** that defines an anonymous lisp function that will act as the predicate. At run-time, the lisp function will be evaluated, and if it returns non-nil, the predicate succeeds. Note that unlike Rhet predicate definitions in the previous item, only one **DefRhetPred** may be defined on a predicate if it is to expand into a lisp function. Note that if the lisp function will have embedded Rhet forms that contain references to the variables in the Argument-Lambda-List, the **DefRhetPred** must be surrounded with the #[and #] special characters to put these references into a single environment, to assure they all refer to the "same" variable. See, for example, 2.19.

5. If the body is empty, the Argument-Lambda-List is simply taken as a declaration form, and the various binding types are recorded. This would be useful, for instance, if one wanted to have specific, but various, constants or lists in the LHS, and thus could not cover all relevant forms of the predicate with a single **DefRhetPred**. Instead, one would use the **DefRhetPred** to make type and usage declarations for the LHS, and then use **Assert-Axioms** to add the various axioms to the KB. For an example of this usage, *see [Miller, 1990a]*. New

The idea behind **DefRhetPred** is to have a somewhat more concise representation (like DEFUN, DEFMACRO, these defining forms expand into the existing Rhet axiom definition forms). It is also a convenient place to put declarations that can be used by the compiler or the Rhet run-time/tracing system to help detect errors and improve efficiency[14]. Further, it is a simple mechanism for redefining groups of related axioms, similar to the incremental nature of DEFUN in lisp. (i.e. the new definition replaces the old, an addition axiom is not just added, leaving the old to be separately deleted as using **Assert-Axioms** would require).

DefRhetPred claims to provide the complete definitions for a predicate. Other axioms or declarations made about the predicate are deleted on re-evaluation of the **DefRhetPred** form. This allows one to simply edit a particular portion of a **DefRhetPred** form and recompile it or reevaluate it to change the Rhet KB, rather than the much more laborious task of either making sure the indexes on the predicates are unique (so a particular one can be deleted), or having to delete all predicates with a particular name, and then reasserting all of them. Furthermore, the declaration style is very similar to lisp's (these are lisp forms, and not available as builtins) to make the syntax easy to remember.

For example:

[14] While currently most such declarations will be ignored, it seems like a good idea to have developed Rhet code get into the habit of using such declarations, if for no other reason than documentation.

```
(5.1)      (DEFRHETPRED P
                  (&Bound ?x*T-U &Any ?y*T-List &Unbound ?z*T-List)
               ;; variables following a &Bound must always bound when
               ;; the predicate is run.  &Any indicates they may or may
               ;; not be bound, and &Unbound indicates they will NEVER
               ;; be bound.  &context may also appear to *set* the
               ;; context the predicate is operative in.  Similarly for
               ;; &forward to define a trigger.  &forward w/o a trigger
               ;; is an implicit :all.

               (DECLARE (OPTIMIZE SPEED SAFETY) ; pass on to compiler
                        MONOTONIC)              ; monotonic declaration
                                                ; means once asserted,
                                                ; it will never be
                                                ; retracted.  (it is an
                                                ; error if it is)

               ;; if the first thing here is an index, we are defining
               ;; the predicate as a series of Rhet rules.

               <1 [FOO ?X ?Z ?Y]
                  [BAR ?Y]

               ;; note that an index separates the rules.

               <2 [MUMBLE ?Z ?X ?Y])

(5.2)      (DEFRHETPRED Q (&LOCAL ?X*T-HUMAN ?Y*T-ATOM)
              (DECLARE (OPTIMIZE SPEED)
                       NON-MONOTONIC)

              ;; First thing here is not an index, so we are defining
              ;; a LISP function as a predicate.  This will create an
              ;; anonymoustfuncompiland the lisp form passed to

              (IF (EQUAL ?Y :TEST)
                  (PROVE [MUMBLE ?X])
                  (ASSERT-Axioms [MUMBLE ?X])))
```

Note that the above functions should automatically expand into the appropriate lower level Assert-Axioms, as well as internal functions for the declarations that are not currently documented, which will not otherwise be available.

Given the above, if I were to eval:

```
(5.3)      (DEFRHETPRED Q (&local ?x ?y)
```

...)

I would NOT redefine Q[15], because the predicates are sensitive to their argument lists (note that ?x was not specified to be of type *T-HUMAN).

Remove-B-Axioms <*Axiom LHS*>
All backward chaining axioms that have a LHS which will unify with the one passed are removed from the KB.

Remove-B-Axioms-By-Index *Index*
Taking the Index as a regular expression, all BC axioms whose Index match, are removed. This index may be either an atom or a string; see 5.1.1.1.

Remove-F-Axioms *Trigger*
Removes all forward chaining axioms with a trigger that unifies with the passed Trigger.

Remove-F-Axioms-By-Index *Index*
As for BC axioms.

Reset-Rhetorical &Optional *Keep-Modes*
Restores system to original unused state. This is loosely the equivalent of cold booting an empty world and loading the system from scratch. If Keep-Modes is non-nil, options (*e.g.* the type assumption mode) are not reset to their default values. Mod

5.2.2 Examining Axioms

List-F-Axioms *Trigger*
Returns a list of all axioms that match the literal trigger. A trigger of [X :all] will return all axioms with trigger head X.

List-B-Axioms <*Axiom LHS*>
All BC axioms whose LHS matches the one passed are returned. A parameter of [X :all] will return all axioms with LHS head X.

List-F-Axioms-By-Index *Index*
be either an atom or a regular expression as per section 5.1.1.1. All FC axioms that match are returned.

List-B-Axioms-By-Index *Index*
Similar to the above, but for BC axioms.

[15] If the DefRhetPred is for a lisp function, then it is not sensitive and would cause a redeclaration.

5.3 Proof We Must

Prove &Rest *Forms* &Key *Proof-Mode Context*

Attempts to prove the list of Forms, and returns a bound solution if one is found. This will be a list of the Forms in the same form as given to **Prove**. By default **Prove** will use the proof mode currently in force (see section 9) which is initially default reasoning mode. The user may specify **:Proof-Mode :Default**, **:Proof-Mode :Complete**, or **:Proof-Mode :Simple** to force the issue. Note that as in 2.18, if the forms passed to **Prove-All** contain variables that are the same between these forms (Rhet forms at top level normally use distinct variables, even if they "print" the same), then the #[and #] characters should be wrapped around the list of the goals to be proven.

Prove-All &Rest *Forms* &Key *Number-of-Proofs Proof-Mode Context*

Does an entire search of the axioms and returns all solutions found. If Number-of-Proofs is given a value of some integer, only the first integer proofs are returned. The value of Proof-Mode is set as in the Prove function. Note that currently if there is an infinite path in the proof tree (*e.g.*, a transitivity axiom) then this function will not return, unless bounded with the **:Number-of-Proofs** keyword. Will return a list of the lists of the forms with their variables bound appropriately, *e.g.*, **(Assert-Axioms [[happy joe]<] [[happy mary]<] [[sad frank]<])**, **(Prove-All [happy ?x] [sad ?y])** returns **(([happy joe] [sad frank]) ([happy mary] [sad frank]))**. Note that as in 2.18, if the forms passed to **Prove-All** contain variables that are the same between these forms (Rhet forms at top level normally use distinct variables, even if they "print" the same), then the #[and #] characters should be wrapped around the list of the goals to be proven.

5.4 Now where did I put that?

Rhet allows the creation of a hierarchy of User Contexts, or UContexts. All functions that specify that they take a context as an argument, actually want RNContexts, as produced by some of these functions[16]. The distinction between a UContext and an RNContext is that the latter is an explicit Rhet Normal Context, which is a combination of a UContext and an implicit or explicit modal operator, e.g. "MB", "SB", or "HBMB". We start with the default Context "T", and if we add something, it goes into this context. But it's RNContext is "SBMB-T" because "SBMB" is the default modal operator. Other combinations of modals and UContexts are possible, what is important to know is that RNContexts are only created when they are needed, since the number of modals that may apply to a context are potentially infinite. To create a new UContexts, say "Foo", is to (implicitly) create a child context for each of the leaf RNContexts. Thus "SB-T" would be the parent of "SB-Foo"[17].

[16] They are described here as RNContexts rather than as Contexts, because a Ucontext is actually a mapping of a user's string to a structure, which can be mapped to one of several RNContexts. There may be Contexts created directly using the internal mechanisms that are not RNContexts, but still allowed. This would be for the programmer who was using Rhet as a programming language, for instance, and wanted a more flexible view of Contexts.

[17] Note that "MB" Contexts are *not* created by creating a new UContext, since the intuitive construction would not give us tree-like context inheritance, or change the existing context inheritance, which Rhet does

Convert-Name-to-UContext *Name*
Given a Name, find the corresponding UContext, if there is one, otherwise it enters the debugger to force the user to supply the name of an existing UContext. If this error processing is unnecessary, or undesired, use UContext-P.

Create-UContext *Name Parent-Name*
Make a new UContext with name Name and parent Parent-Name. It will be initially devoid of children and RNContexts.

Destroy-UContext *Name*
Forcibly remove a UContext and all its descendants, including all contained RNContexts.

Operator *Name* &Key *Ucontext*
Given the name of a modal operator, Operator, tries to determine an associated RNContext (meant to be used as the argument to the :Context keyword). Normally the Ucontext will be the default one, but the user may specify a particular Ucontext, with the keyword. That is, to make the default context be "Foo" and the default operator "SBHB", one would do:

```
(5.4)    (Setf *Default-Context*
               (Operator "SBHB" :Ucontext (Convert-Name-to-UContext "Foo")))
```

UContext *Name*
Given the name of a UContext, tries to determine an associated RNContext (meant to be used as the argument to the :Context keyword). Note that the default modal operator in use at the time will be used in determining the RNContext. If you want to specify an operator as well, use the Operator function, described above.

UContexts
Return a tree of all the UContexts known to the system[18].

UContext-P *Name*
Is Name a UContext? Returns the UContext if it is, else Nil. If you would like to assume that you will always get a valid UContext returned, use Convert-Name-to-UContext, instead.

5.5 Unity

Rhet allows the user to call the unifier directly. Also see section 5.7.4 for those functions dealing with variable arity predicates.

---

not allow.

[18]For a list of all RNContexts known to the system, including some internal ones used by Rhet, call (Contexts).

Unify *Form1 Form2*

This returns several values: the first is a success indication, and is non-Nil if unification between the two forms succeeds. The succeeding values are the Rhet variables as they were bound by the unification (Rhet destructively modifies the passed variables to indicate their bindings). Unify ignores equality assertions.

E-Unify *Form1 Form2*

Similar to Unify, but this call does E-unification, which Rhet normally does by default during proofs.

5.6 Consing Forms

New

Normally, if one has set one's readtable appropriately[19], one can freely embed Rhet forms within lisp expressions. Given a usable implementation of Common Lisp, these will self-evaluate, and not need to be quoted. It may, however, be desirable to CONS Rhet forms on the fly. Here are a list of some useful functions and examples. More detailed information might be found in [Miller, 1990b].

Create-Rvariable *Pretty-Name* &Optional (Type *T-U-ITYPE-STRUCT*)

A function to construct variables. The Pretty-name is a string that will be the printed appearance of the variable. *As such, it should begin with the character "?".*

Cons-Rhet-Axiom *LHS* &Rest *RHS*

Return a standard Rhet bc-axiom (unasserted) given a list representation, *e.g.*

```
(Let ((var-x (Create-Rvariable "?X" (Make-I-Type 'T-FOOBAR))))
  (Cons-Rhet-Axiom (Cons-Rhet-Form 'P var-x)
                   (Cons-Rhet-Form 'Q var-x)
                   (Cons-Rhet-Form 'R var-x)))
[[P ?x*T-Foobar] < [Q ?x*T-Foobar] [R ?x*T-Foobar]]
```

Cons-Rhet-Form *Head* &Rest *Arglist*

Return a standard Rhet form given a list representation, *e.g.*(Cons-Rhet-Form 'P 'A) returns [P A], (Cons-Rhet-Form 'P '(A B)) returns [P (A B)]. If the head looks like a builtin, it will be handled appropriately. Rhet variables should be created with **Create-Rvariable** and EQness will be preserved between calls to this function if the same Rvariable structure is passed.

Make-I-Type *<Type Symbol or List>* &Optional *Permissive*

Converts the passed lisp symbol into an Itype-Struct. If a list is passed, it is treated as a type specification as would appear after the asterisk on a variable. This function is typically used, for example, by builtins that expect an argument to be a type specification. If Permissive is NIL, the default, return NIL if any component type is undeclared. It is usually unnecessary to convert a predefined type into an **Itype-Struct**, because they are

[19]E.g. by setting the Syntax attribute to "Rhet", as per 8.2.3

all available on globals. *T-U-ITYPE-STRUCT*, for instance, holds the Itype-Struct for type *T-U. All of the types in table 2.3 are available on similar variables, e.g. *T-Orthodox-Set would be available in *T-ORTHODOX-SET-ITYPE-STRUCT*.

5.7 The Rhet/Lisp Interface

So far, we have seen how the various Rhet facilities can be invoked from within Lisp. This section explains how Lisp facilities can be used within Rhet.

5.7.1 Calling a Lisp predicate directly

It is possible to make the proof of some Rhet form directly depend on a Lisp predicate. This is done with the following function:

[Call <Lisp Expression>]

This predicate simply invokes the Lisp expression, and if it returns non-Nil, it succeeds. For more detail, see 4.3. Note that the Lisp expression is expected to be a list, not a function, so Rhet may replace variables mentioned with their bindings (this means the user function doesn't need to know how to handle bound variables).

5.7.2 Assigning Lisp Values to Rhet Variables

There is a simple mechanism for binding a Rhet variable to an arbitrary Lisp value. This is accomplished by using the built-in predicates:

[Setvalue <Variable-List> <Lisp expression>]

This evaluates the <Lisp expression> as a Lisp program and binds the results to the Rhet variables specified. If the variable is already bound, SETVALUE will fail. If the Lisp expression returns multiple values, each value is bound to each variable in the variable list in turn. Extra variables specified are bound to NIL. Extra values returned are ignored. Note that this is also a Rhet predicate, and may appear inside of axioms[20]. For example, calling:

```
(5.5)    [Setvalue '(?x*T-ATOM ?y*T-ATOM) '(VALUES ':A ':B)]
```

would bind ?x to :A and ?y to :B. Similarly `[Set-Setvalue]` works like Setvalue, but binds the variables to Sets instead of lists.

[20]It is, frankly, much more useful as a predicate, however, it does allow user functions called by Rhet via `[Call]` or Declare-Lispfn to handle unbound variables.

[Genvalue <Variable-List> <Lisp expression>**]**

This is the same as [Setvalue] except that the Lisp expression is expected to return a list of values. Thevariable will be bound to the first value, and if the proof backtracks to this point, to the succeeding values one at a time. If the Lisp expression returns multiple values, each value is assumed to be a list, and each value returned is associated with a variable in the variable-list. Note that this is also a Rhet predicate, and may appear inside of axioms. A slightly modified example 5.5 would give us something like:

```
(5.6)    [Genvalue '(?x*T-ATOM ?y*T-ATOM)
                   '(Values '(:A :B) '(:C :D))]
```

which, after parsing through the Lisp multiple values, would bind ?x to :A and ?y to :C, then if backtracking happened, would next bind ?x to :B and ?y to :D.

[Set-Setvalue <variable / list> <Lisp-Expression>**]**

New

Sets the Rhet variable <variable> to the value of the Lisp expression <Lisp-Expression> as a Set, rather than a list. Any logic variables in <Lisp-expression> are replaced by their logic bindings before Lisp evaluation. If the Lisp expression returns more than one value (in the Common Lisp sense), then the values are assigned to each variable in the list in turn. Extra variables in the list are bound to Nil. Too few variables cause the extra values to be discarded. For example: `[Set-Setvalue ?x*t-set (quote (a b c))]` succeeds binding ?x to {`a b c`}.

5.7.3 Lisp Functions as Predicate Names

Occasionally it is useful to let a predicate name be a Lisp function that gets called instead of letting Rhet prove the form as usual. These special Lisp functions will be applied to a frozen form of the arguments coded, rather that to the actual rhet forms. They cannot support backtracking or side effects, so the need for either should be avoided. They receive their argument list from Rhet with all bound variables replaced by their values. More information is available in [Miller, 1990b]; there are many possible issues that a user may need to be aware of to code any but a very simple Lispfn.

To declare such a Lisp function to Rhet use:

Declare-Lispfn <*Name*> *Query-Function-Symbol* &Optional *Assert-Function-Symbol Type-Declarations*

Declares to Rhet that Name is not a predicate but a Lisp function; Rhet will recognize those <Name>s as calls to Lisp functions. If the Reasoner is attempting to prove an axiom that has been declared a Lisp function, it will call the Query-Function-Symbol (passed as a symbol to allow it to be incrementally recompiled). If it attempting to add a predicate to the KB whose head is declared with an associated Assert-Function, it will call the Assert-Function-Symbol rather than add it[21]. Lisp Query-Functions

[21] It will still forward chain as if it had added it, although it may not detect a loop!

should only return "t" or "nil" which will be interpreted as true and false respectively. The optional Type-Declarations are a list of symbols representing the types of the arguments expected by the lispfn. This will be used for runtime debugging interaction, and as a hint to the compiler. See also [Miller, 1990b] for information on lisp functions that are available to deal with rhet objects that may be handed as arguments to a lispfn.

For example, assume we enter the following[22]:

```
(defun check (&rest x)
    (terpri)
    (princ "in check, args are:")
    (princ (Mapcar #'real-rhet-object x))
    t)
```

(5.7) (Declare-Lispfn 'check 'check)

(5.8) (Assert-Axioms [[P ?x ?y] < [check ?x ?y]])

Then if we call

(5.9) (Prove [P A B])

the Lisp function check is called resulting in the output:

`in check, args are:` (A B).

Since check returns a non nil answer, the Lisp call is treated as a success.

Note that the package of the assert and predicate symbols are significant, but the package of the Name of the lispfn is not. So, there may only be one lispfn by a particular name; Rhet does not support packages within forms.

Lispfns cannot backtrack, and therefore cannot bind variables. Neither can they have side-effects you want Rhet to be able to undo. If you need either of these things, you need to define your own builtin. See [Miller, 1990b], the Rhet programming guide, for more details on how to go about doing this. If a lispfn is called or asserted with an unbound variable embedded in it's argument list, Rhet will signal an error.

A few useful functions for manipulating argument lists within Lisp are:

Rvariable-P <*Any Lisp Object*>
Returns NIL if the object is not a rhet variable structure.

Rvariable-Pretty-Name <*Variable*>
Returns the variable name.

[22] Real-Rhet-Object is documented in [Miller, 1990b]

Rvariable-Type < *Variable*>

Returns the type of the Rhet variable.

Note that both of the above functions expect to operate on actual variable structures, as Rhet would give to a unsuspecting lisp function if a variable is unbound, or as the reader would produce as the value of typing "?x". Other useful functions can be found in the Programmer's Guide.

5.7.4 Using Lists in Rhet

Rhet is embedded in Lisp, and one can use the Lisp list facility directly. The Rhet unifier normally operates on an internal structure called a Form. This is what an expression such as `[Father-of Mary]` is turned into by the reader. But unification will work on Forms or Lists. Lists, however, do not unify with Forms; they only unify with other Lists. Further, few builtins will work with Lists, normally Forms are expected as arguments.

The unifier will handle the dot operator appropriately anywhere it is syntactically legal. Thus the following pairs of terms unify with the most general unifier shown:

(a b c) (a ?x ?y)	with m.g.u. ?x/b, ?y/c
(a b c) (a . ?x)	with m.g.u. ?x/(b c)
(a b c) (?x . ?y)	with m.g.u. ?x/a, ?y/(b c)
(a b c) (a ?x . ?y)	with m.g.u. ?x/b, ?y/(c)
(a b) (a ?x . ?y)	with m.g.u. ?x/b ?y/nil
(a) (a ?x . ?y)	does not unify.
(a b) (?x)	does not unify. (?x) only matches lists of length 1.

Form unification is also allowed with varying arity predicates.[23] The main difference is that rather than the Lisp dot operator, the lambda form &Rest is used instead. &Rest binds all the remaining parameters of a function term to the Rhet variable following the declaration as a List of Forms. It is semantically equivalent to a dot operator, but that is only legal in Lists and not Forms. This List can be decomposed in the usual manner, as described above[24].

Consider the definition of the predicate `[or*]` that is true if any of its arguments is true[25]:

```
[[or* ?x &rest ?y*T-List] < ?x] ; or* is true if the first argument is true
[[or* ?x &rest ?y*T-List] < [or** ?y]] ; or* is true if or* of all but the
```

[23]Even with a var in the predicate name position!

[24]I will note again that by default the type of a variable is T-U, when it is present in a form, which cannot be bound to a List. Therefore a form such as `[?x &rest ?y]` will never unify with anything. The user should instead use `[?x &rest ?y*T-LIST]` or `[?x &rest ?y*T-Lisp]`. The default for a variable inside of a list is T-Lisp.

[25]Actually, the builtin `[Or]` would accomplish this, but for illustrative purposes. . .

```
                                          ;first argument is true. Note or** usage.
[[or** (?x*T-U . ?y)] < [or* ?x]]         ; need or** to handle LIST vs. FORM syntax
[[or** (?x*T-U . ?y)] < [or** ?y]] ; note typing on variables.
```

Thus the call with no arguments, `[or* ]`, always fails and each of `[or* [A]]`, `[or* [B] [A]]`, and `[or* [B] [A] [C]]` succeeds if `[A]` is provable. The clumsiness of the two different representations[26] for LISTs and FORMs is what drives us to have to write both or* and or**; this is only a problem for varying arity predicates (those that use **&Rest**).

The two direct interfaces to the unifier from Lisp, **Unify** and **E-Unify** (see section 5.5) would as expected work with varying arity predicates.

For example: Lets say we know that `[A]`, `[P Q]` and `[B]` are all declared equal.

- `[A ?x]` and `[B C]` will **E-Unify** with m.g.u. ?x/`[C]` but will not Unify.
- `[A &rest ?y*T-List]` and `[A B C D]` will both **Unify** and **E-Unify** with m.g.u. ?y/(`[B] [C] [D]`).
- `[P ?x]` and `[B]` will not **Unify**, but will **E-Unify** with m.g.u. ?x/`[Any ?x [EQ? [P ?x] B]]`. This is intuitive when you remember that there may be more than one binding of ?x that satisfies the equality, and we would not want to prematurely bind it.
- `[P ?x]` and `[C]` would neither **Unify** nor **E-Unify**. It will not E-Unify because there is *currently* no binding of ?x possible that could make `[P ?x]` equal to `[C]`.

5.7.5 Manipulating Answers from Rhet

Since Rhet deals with it's own ideas of how variable values are found, some Lisp functions are included to allow user code to more easily manipulate the results from certain functions.

Get-Binding <*Variable*>
: Returns the binding for the variable. This is an actual Variable Structure, as is returned by the reader on "?x" or can be passed by Rhet to Lisp functions.

List-Forward-Chained-Facts
: This function returns a list of all axioms that have been asserted via forward chaining since the last call to this function.

5.8 Equality

This subject is discussed in further detail in the tutorial.

The Rhet predicate **Add-EQ** is also available as a Lisp function:

[26] Horne had only one, but lost expressivity

Add-EQ *Term1 Term2* &Key *Context*

Both terms must be fully grounded. This Lisp form will add the equality specified. Note that the context the equality is added in may be explicitly specified. Note that this function is also available as a Rhet predicate.

There are three Lisp functions for examining the equality assertions:

Equivclass <*Ground Term*> &Key *Context*

Returns a list of all ground terms equal to the <Ground Term> in the provided Context.

Equivclass-V *Term* &Key *Context*

Returns a list of all terms that could be equal to the term followed by variable binding information, in the provided Context.

Primary <*Ground Term*> &Key (Context *DEFAULT-CONTEXT*)

Returns the fact structure of the primary instance of the class the passed term is a member of. This is usually the simplest member[27] of the class, *e.g.* if we know

(5.10) `[Add-EQ [Mother-of John] [Francine]]`

then

(5.11) (Primary [Mother-of John])

will return `[Francine]`. Note that the system decides when it is appropriate to update the primary object for a canonical class, it is not something the user can declare.

Print-FN-Term-Pretty

New

Controls pretty printing of function terms. If non-nil (the default) the primary instance of a canonical class will be printed for any function term, if false, the exact function term will be printed. So for instance, if at the top level one were to write

```
Rhet-> (Add-EQ [A] [B])
:EQ
Rhet-> [P A B]
[P B B]
Rhet->
```

because the equality allows the system to print out `[A]` as `[B]`. This is the reason the :Candidate-Primary option on Dtype, Utype, and Itype may be important: the user may specify a non-atomic term may be used as the primary for a class. Note that Rhet assumes any atomic term is a legal primary, and it is non-defined among the set of possible primaries, which term will become the *actual* primary.

[27] That is, the function term with the least number of arguments.

5.9 Inequality

The horn predicate `[Add-InEQ]` is also available as a Lisp function:

Add-InEQ *Term1 Term2* &Key *Context*
Both terms must be fully grounded. This Lisp form will add the inequality specified. Note that the context the inequality is added in may be explicitly specified. Note that this function is also available as a Rhet predicate.

There is also a Lisp function for examining inequalities.

InEquivclass *<Ground Term>* &Key *Context*
Returns a list of all ground terms that could not be made equal to the <Ground Term> in the provided Context. This may be because the terms are of disjoint types, distinguished subtypes of the same type, or declared to be unequal via Add-InEQ. Note that this is an expensive operator and should only be used in exception circumstances.

Chapter 6

Types

6.1 Adding Type Information

The following Lisp functions are supported for adding type information. Note that Rhet does not support type manipulation during proofs, so these functions are not available as builtins.

Tdisjoint *TypeName**
: Asserts that all the types mentioned are pairwise disjoint.

Tname-Intersect *Newtype TypeName**
: Asserts that the intersection of all TypeName* is Newtype. *E.g.*, (Tname-Intersect 'Woman 'Human 'Female) says that objects that are both subtypes of type Human and type Female are of type Woman. Note that if more than two types are present in TypeName*, Tname-Intersect may create new named intersections in order to appropriately build the type table. Warnings are given when this happens.

Toverlap *TypeName**
: Asserts that all TypeName* overlap, but that their intersection is unnamed. This is not needed when type assumption mode is active, since all types not known to be disjoint will be assumed to overlap. See 9.2.

Tsubtype *Supertype Subtype**
: Asserts that each type in Subtype* is a subclass of the Supertype,*e.g.*, (TSUBTYPE 'ANIMAL 'CAT) asserts that CAT is a subclass of ANIMAL.

Txsubtype *Super-Type Type**
: Asserts that all type* are a partition of Super-Type, *i.e.*, they are all subtypes of Super-Type, that all Type* are pairwise disjoint, and that the union of all Type* is equivalent to Super-Type[1].

[1]As of V.17.9, Rhet only recognizes Txsubtype as an abbreviation for Tsubtype and pairwise Tdisjoint declarations. Eventually, we hope to be able to take advantage of the additional information Txsubtype implies.

The system that adds a TYPE declaration and its implications to the matrix first checks that the statement is consistent. If the statement contains an inconsistency, an error message is printed and no information is added to the matrix. For example, if one adds **(Tdisjoint 'cats 'dogs)** and then adds **(Tsubtype 'cats 'dogs)**, an error message will be given and information in the second declaration will not be added to the matrix.

In order for the matrix system to derive all implied information, **Itype** assertions should be added after **Tsubtype**, **Txsubtype**, **Tdisjoint**, and **Tname-Intersect** assertions. In fact, the entire type table should be in place before any **Assert-Axioms** are done. The type table is expected to remain constant during a proof, therefore declaring any of the above to be Lisp functions to Rhet, and calling them from inside of an axiom is not recommended.

Type restrictions on the arguments to a function term, and on the type of the function term itself, are declared using the form:

Declare-FN-Type *Fn-Atom* (Type1 ...TypeN TypeN+1) &rest *Function-Specs*

Asserts that Fn-Atom is the name of a function that takes arguments of the types Type1,...,TypeN and describes objects of type TypeN+1. For example, **(declare-fn-type 'ADD '(T-Odd T-Odd T-Even))** declares a two-place function ADD, such that when both arguments are of type T-Odd, it will produce an object of type T-Even. Other examples:

- (Declare-FN-Type 'Father '(T-Human T-Man))
- (Declare-FN-Type 'Spouse '(T-Human T-Human) '(T-Man T-Woman) '(T-Woman T-Man))
- (Declare-FN-Type 'Parent '(T-Human T-Human))

Function-Specs is an arbitrary number of *Function-Spec-Clauses*, where each Function-Spec-Clause has the form (Type1 Type2 ...Type n+1) In general we use 'function specification' to refer to typing information about function terms. Informally this means that if the i-th argument is of type-i (i = 1,...,n) then the result is of type-n+1. If a list of functions appears as the first argument as in the third example above, the same specification is declared for all those functions. If there was an old declaration for function-atom, it will be overridden. **Declare-FN-Type** returns all the arguments as they are given.

Type specification is actually associated with p-names (string) of function-atoms thus it doesn't matter in which context a user makes a declaration and a declaration is effective for all the contexts.

Note that the macro **DefRhetFun** is a somewhat nicer interface to this function, and is recommended to be used instead of **Declare-FN-Type** directly from the top level.

The declarations to Declare-FN-Type will also be inverted. That is, given the above example for Spouse, if a term `[Spouse ?x]` is found and known to be of type T-Woman, ?x will be derived to be of type T-Man. Had we instead declared:

(Declare-FN-Type 'Spouse '(T-Human T-Human) '(T-Male T-Female) '(T-Female T-Male))

which seems a reasonable declaration, we run into two problems:

1. This reverse typing cannot be done, because no result is of type T-Woman. While in theory we could derive such information, the current implementation does not do so.
2. When we specify types for Declare-FN-Type we must assure that it works for all subtypes, and the types are contained in a hierarchy. That is, in this restatement, T-Male is not a subtype of T-Human, nor is T-Human a subtype of T-Male. It is an error to give **Declare-FN-Type** arguments that are not specializations of the first argument. This example could be "fixed" by making the first declaration (T-U T-U), but in general we may not want to do this so we can trap bad arguments to the Spouse function, *e.g.* an argument type that is not a T-Human.

So, the rules of thumb are:

1. The most general argument type should come first, and all arguments to the typed function must be subsets of this type.[2]
2. Until Rhet is smarter, there should be a result form for each possible result type you expect the function to return so inversion works.
3. Each declaration must be independently true. That is, again for Spouse, we have said that if the argument is of type T-Human, *or a subtype of T-Human*, then the result will be of type T-Human *or a subtype*. Thus the following would be an error:

```
(6.1)     (Declare-FN-Type 'Foo
                           '(T-Human T-Human T-Lisp)
                           '(T-Human T-Man T-List)
                           '(T-Man T-Human T-List)
                           '(T-Man T-Man T-Atom))
```

   because the last declaration is in conflict with the second and third; if both arguments are of type T-Man, then the result is of type T-List by both the second and third entry, but of the *disjoint* type T-Atom by the last. In general, the more specific the arguments, the more specific the result type should be, and always a subtype of the less general ones.

Note that **Declare-FN-Type** will check any previously type constrained variables to see if they now simplify (due to the new information). New

A user can access/modify function specifications using the following functions. The syntax of each function is exactly like declare-fn-type — each allows one or a list of function atoms as argument (except 'look-up-fn-type' which takes one function-atom only) and none of the arguments is evaluated so a user doesn't have to quote any argument. They pretty much do what you expect them to do and all of the modification functions (*i.e.* all except **Look-Up-FN-Type**) return a list of arguments as given.

[2] Rhet will give a warning (upon use) if this rule is violated; Rhet is free to coerce an axiom or term of the form `[[P ?x ?y] < [EQ? [Spouse ?x] ?y]]` into one where ?x and ?y are both typed T-Human (given the example definition for Spouse) because the declaration said that T-Human was the most general type Spouse would accept, not T-U.

Add-FN-Type *Function-Atom/List-Of-Function-Atoms* &Rest *Function-Spec*

New

Function-Spec is added to (each) Function-Atom. Note that **Add-FN-Type** will check any previously type constrained variables to see if they now simplify (due to the new information).

For example, given that we have asserted:

(6.2) (Declare-FN-Type 'Spouse '(T-Human T-Human) '(Male Female) '(Female Male))

to declare that the Spouse function will take arguments of type T-Human and itself be a subtype of T-Human, and further that if the argument is a subtype of Male, then it is a subtype of Female, *etc.*,

(6.3) (Add-FN-Type 'Spouse '(T-Academic T-Academic))

would further constrain this type such that academics would only marry academics.

Clear-All-FN-Type

Removes all the function specifications declared so far.

Delete-FN-Type *Function-Atom/List-Of-Function-Atoms* &Rest *Function-Spec*

Each clause appearing in Function-Spec is deleted from the specification of (each) Function-Atom if it exists. We could use, for instance,

(6.4) (Delete-FN-Type 'Spouse '(T-Academic T-Academic))

to undo our annotation of 6.2 by 6.3.

Get-All-FN-Type

Returns all the function specifications declared so far.

Look-Up-FN-Type *Function-Atom*

Returns a function specification for function-atom. Returns Nil if none has been declared.

Remove-FN-Type-Def &Rest *Function-Atoms*

Removes the function specification associated with (each) function-atom.

DefRhetFun *Function-Name ((Most-General-Arg-Type1 ... Most-General-Arg-TypeN) Most-General-Result-Type)* &Body Body

A macro to make top level definition of function terms more painless. It claims to provide the complete definitions for a function term. Other declarations made about the predicate are deleted on re-evaluation of the DefRhetFun form. This allows one to simply edit a particular portion of a DefRhetFun form and recompile it or reevaluate it to change the declarations, rather than the more laborious task of using Remove-FN-Type-Def, then using Declare-FN-Type to redefine it. The declaration style is intentionally very similar to lisp's (these are lisp forms, and not available as builtins) to make the syntax easy to remember. Not in V17.9

The body may consist of the following:

1. If the first object is a string, it is considered a documentation string.
2. If the first object after the optional documentation string is a Declare form, it is treated like CL Declare forms, that is, as instructions to the implementation that may assist in generation code or debugging information, but not change the language construct itself (the correctness of the program and the results that are derivable from a correct program are independent of declarations).
3. After either of these optional constructions, a keyword is expected. This may be one of the following:

   :Alternates which is followed by one or more alternative type definitions, as accepted by Declare-FN-Type. These will be evaluated, so should be quoted if necessary. All of the types must be subtypes of the respective Most-General-Arg-Type or Most-General-Result-Type, and follow all the other rules of typing that Declare-FN-Type requires.

An example:

(6.5)
```
(DEFRHETFUN MOTHER-OF ((T-SEXED-ANIMAL) T-FEMALE)
    ;; most general allowed types first.  If safety is on,
    ;; and Father-of is used on an argument not a subtype
    ;; of T-Sexed-Animal, we get an error.
    (DECLARE (OPTIMIZE SAFETY))

    :ALTERNATES ; alternative types
    '((T-HUMAN) T-WOMAN)
    '((T-DOG) T-BITCH)
    ;; what we'd REALLY like (but can't yet express) is
    ;; something like:
    '((?X*T-SEXED-ANIMAL)
      [ANY ?Y*FEMALE (TYPE-COMPATIBLEP
                        (GET-TYPE-OBJECT ?Y)
                        (GET-TYPE-OBJECT ?X))]))
```

Note that the above functions should automatically expand into the appropriate lower level DECLARE-FN-TYPEs, as well as internal functions for the declarations that are not currently documented, which will not otherwise be available.

Given the above, if I were to eval:

```
(DEFRHETFUN MOTHER-OF ((T-SEXED-ANIMAL) T-FEMALE)
...)
```

I would redefine the mother-of function, independent of the arglist I used.

Examples and further discussion on function typing are found in the system overview in the tutorial.

6.2 Lisp Interface to Type System

There is a set of Lisp functions to access and use the type system independently of Rhet. The most important function returns the type of an arbitrary Rhet term:

Get-Type-Object *Term*
: Given any Rhet term, this function returns the most specific type of that term. If the term contains one or more variables, it returns the most specific type that includes every instantiation of the term. This may or may not depend on a functional type for the object being defined.

Matrix-Relation *Type1 Type2*
: Returns the information that is stored in the matrix for the relationship between the two types.

Type-Compatiblep *Type1 Type2*
: Takes any two types and returns T if the types are identical, or if Type1 is a proper subtype of Type2[3].

Type-EQ *Type1 Type2*
: Returns T if Type1 is equivalent to Type2. For example, if we have defined types B and C as exclusively partitioning type A, then we would expect (Type-EQ 'B '(A-C)) to succeed. The type reasoning supported by the type subsystem may not be general enough to derive that two types are Type-EQ even though they are. For example, if several partitions of a type are defined, and other constraints on these partitions are known, the type subsystem may not completely derive other relationships between these partition types.

[3]This is essentially a nice user interface to the internal function Typecompat, which is one of the primary interfaces between Rhet's unifier and the type subsystem.

Type-EQL *Type1 Type2*

Returns T if Type1 and Type2 overlap with no named elements NOT in common, *i.e.* $Type1 \cap Type2 = Type1$. By named element, we mean type, *e.g.* if T1 and T2 are defined to overlap, but T3 is a subtype of T1 and not of T2, then T1 and T2 are not Type-EQL. Not in V17.9

Type-Exclusivep *Type1 Type2*

Succeeds if *by definition* $Type1 \cap Type2 = \emptyset$.

Type-Info *Type*

Returns a list giving the relationship between the given type and every other type in the system, of the form:

((rel type1)(rel type2)...)

Type-Intersectp *Type1 Type2*

Succeeds if $Type1 \cap Type2$ has some named instance, that is, a named intersection. Note that this is stronger than (NOT (Type-Exclusivep T1 T2)).

Type-Subtype *Type* &Key *Recursive*

Returns a list of the immediate subtypes of a type. If the Recursive option is supplied and non-nil, this function returns the closure of all immediate subtypes.

Type-Subtypep *Type1 Type2*

Succeeds if $Type1 \subset Type2$. This is distinct from Type-Compatiblep, in that we must find some defined or inferred type that is a subset of Type2 and not of Type1.

Type-Supertype *Type* &Key *Recursive*

This returns the immediate supertypes of the type as defined, if known. If the Recursive option is supplied and non-nil, this function returns the closure of all immediate supertypes.

Types

Returns a list of all types known in the system.

Table 6.1 indicates the possible relationships between types:

6.2.1 Type Compatibility and an Example

Using the axioms above, Rhet can compute the compatibility of two terms efficiently, at least if neither term is involved in a type calculus. Types are compatible if one is a subtype of the other or if they overlap. Overlaps occur in two ways: named or unnamed. A named overlap results from an Tname-Intersect assertion; an unnamed overlap can be implied from a Toverlap assertion. The unification of two typed variables may result in a variable of a complex type of the form *(type1 type2) indicating the intersection of the two types. This new type is recognized in the proof as a new type. For example, suppose we have the assertions:

:SUBSET	a subset relation holds between the two types.
:SUPERSET	a superset relation holds.
:NONDISJOINT	the types intersect but the overlap is not named.
:EQUAL	the types are identical.
:PARTITION	(Unimplemented as of Version 17.6) for entry [a b], b is a subset of a, and for all other subsets of a there is no overlap (part of a cover of a). Note that [b a] will be of type :subset, and :partition implies superset.
:DISJOINT	if [a b] do not intersect
:UNKNOWN	if the relationship is undefined, and could not be derived.
Symbol	(other than the above) the item on the list is the name of the intersection of the given types.

Table 6.1: Relationships between Rhet types in the Type Table.

(6.6) (Tsubtype 'T-Anything 'T-Cars 'T-Person)

(6.7) (Tsubtype 'T-Cars 'T-Ford 'T-Small-Cars)

(6.8) (Tsubtype 'T-Person 'T-Student 'T-Worker)

(6.9) (Toverlap 'T-Student 'T-Worker)

(6.10) (Tname-Intersect 'T-Pintos 'T-Ford 'T-Small-Cars)

(6.11) (Utype 'T-Worker [JOHN])

(6.12) (Utype 'T-Student [JOHN])

(6.13) [[want ?x*T-Person ?g*T-Ford] ¡ [efficient ?g*T-Ford]
[wealthy ?x*T-Person]]

(6.14) `[[efficient ?f*T-Small-Cars] <]`

(6.15) `[[wealthy ?d*T-Worker] <]`

We could then attempt to prove

(6.16) `[want ?f*T-Student ?d*T-Ford]`

and we would get

(6.17) `[want ?r*(T-Student T-Worker) ?u*T-Pintos]`,

T-Pintos being a named overlap while the intersection of the types T-Student and T-Worker is derived by the prover.

6.3 Structured Types

Structured types as a special form of Rhet type. They act just like normal Rhet types, but have additional properties associated with them. In a gross sense, they are very much like flavors in Lisp, but rather than each slot having a value, it is used for equality. This has advantages and disadvantages. The principle disadvantage is that a slot cannot be said to really have a "value", and one entry in the equivalence class is treated by Rhet the same as any other. This means that one can't really deal with determining if a "value" has been found yet for a slot except by having your application recognize appropriate members of the equivalence class specially. The principle advantage is in fact related to this problem: we can talk about the roles of several different objects as being "equal", without knowing what their "value" is. In fact, this notion of equality is stronger than this; it is sort of a Lisp EQ vs. EQUALP distinction: some systems allow us to see if the contents of two distinct cells happen to be the same, while Rhet deals with equality on a more fundamental level: the two slots can be DEFINED to be the same. This is not merely sharing a cell, it is a logical relationship using Add-EQ that can be independently reasoned about.

6.3.1 Defining Roles and Relations in the Type Hierarchy

Define-Subtype *Type Supertype* &Key *(Roles* (Rolename Type)**)* *(Relations* (Relationname Relation-Forms*)**)* *Constraints Initializations*

Defines Type as a subtype of Supertype, defines the indicated type restricted roles for the new type, and asserts or notes the Constraints (any Rhet predicates, including equality constraints). In addition, Type inherits any roles from Supertype. An inherited role may be redefined only if its new type restriction is a subtype of the inherited type restriction, *e.g.*,

(6.18) (Define-Subtype 'T-ACTION 'T-U :roles '((R-ACTOR T-ANIM)))

defines T-ACTION to be a subtype of T-U, with a role R-ACTOR defined and restricted to be of type T-ANIM. This is roughly equivalent to adding:

(6.19) (Tsubtype 'T-U 'T-ACTION)

and defining f-actor by

```
(6.20)   (Declare-FN-Type 'f-actor '(T-ACTION) 'T-ANIM).
```

In addition, **Define-Subtype** sets up some internal data structures to maintain the role inheritance in an efficient manner. **Define-Subtype** can also be used to declare that certain roles will be constrained, *e.g.* with the above **Define-Subtype** we could now assert:

```
(6.21)   (Define-Subtype 'T-Self-Action 'T-Action
                         :Roles '((R-Action-Object T-U))
                         :Constraints '([EQ? [F-Actor ?self]
                                             [F-Action-Object ?self]]))
```

Note the use of the variable **?self** as a stand in for the instance being created of this type. In general, one might need to have placeholder variables, as in the following example:

```
(6.22)   (Define-Subtype 'T-Foo-Action 'T-Self-Action
                         :Roles '((R-Foo-Object T-U))
                         :Constraints '([Foo-P [F-Foo-Object ?self]
                                             [C-Mumble [F-Actor ?self]
                                                       ?z*T-Mumble-Role-2]]))
```

This allows matching of the variable to the instance the constructor builds (since it is unused). Only the term **?self** is treated specially by the system.

Initializations are a list of Rhet forms that will be proved at instance creation time. Normally, these will be builtins with side effects. A single variable which can be unified with the instance must appear in the initializations, and at instance creation time, the initializations forms will each be deterministically proved with this variable bound to the instance being created.

There are two basic properties for constraints: foldability and monotonicity. To be *foldable* implies that Rhet can simply assert it to be true and thus achieve the constraint. An example of a foldable constraint would be that one slot is not equal to another. Rhet can use the `[Add-Ineq]` assertion to achieve this sort of constraint. To be *monotonic* implies that once the constraint has been proven to hold, it is guaranteed to continue to hold. Our inequality assertion is an example of this. A call to a lispfn, such as one that checks that the current time is 4pm would be an example of a constraint that does not have this property.

Thus there are four cases of constraint types that need to be considered:

Foldable, Non-Monotonic which are forms that can be undone, but it is desirable to fold their effect in immediately. Rhet can tell that a predicate is Foldable and Non-Monotonic on the basis of declarations made via the **DefRhetPred**

function. Also, this is the default type of constraint. Constraints that are foldable means that their effect may be asserted. Non-monotonic means that once they hold they do not necessarily continue to hold. Currently, Rhet will assert F/NM constraints, but will never test for them, so they are the equivalent of F/M constraints. For example, to claim that constraint `[Foo-p [R-Slot1 ?x*T-Being-Defined]]` is F/NM means that the system will simply Assert this to be true, without binding the variable, *i.e.* do `[Assert-Axioms [[Foo-p [R-Slot1 ?x*T-Being-Defined]] <]]`.

Foldable, Monotonic *e.g.* equality which once done cannot be undone, and are always valid. FC works for these. User predicates that are monotonic would work here too: once we assert them as true we no longer have to worry about them. In some sense, this is equivalent to the Initializations field, except for a small matter of semantics. Where on initializations we would write **Add-EQ**, on constraints we write `[EQ?]`, since here we are trying to describe a set of forms that should be provable, rather than a set of forms that should be asserted.

Non-Foldable, Non-Monotonic These would include many calls to user lisp functions, and builtins such as `[Unless]`, `[Forall]`, and many other builtins. If the user is using a negation-by-failure semantic for a predicate, it is by definition non-foldable, because we cannot assert its failure, and since it is possible to consistently add it later (or assume it, see, for instance, the example in [Miller, 1990a], dealing with the ITALIAN predicate) it is non-monotonic. Mod

Non-Foldable, Monotonic These would include certain builtins that once true will remain true, but cannot be "asserted" to be true in the sense that EQ or a user predicate could. Monotonic lispfns would come under this header, for example, as well. E.g.

(6.23) `[Parent-of [F-Object ?x] [F-Actor ?x]]`

At the surface, this LOOKS like something we can assert to be true. The problem is that if we know that Jim and Sue are Mary's parents, and for this instance we make the object Mary but the actor Bill, then Bill becomes one of Mary's parents! Either we have to have a rule which states that not only Jim and Sue Mary's parents, but Mary has no other parents, or we have to have declared Parent-of to be Non-Foldable, that is, NOT something we want to ASSERT to be true, but something we want to TEST and be SURE it is true. We may not want to write the rule like this (which would confuse us with F/M), so let us instead write something like

(6.24) [EQ? [Any ?x*t-human [Parent-of [F-Object ?y] ?x]]
[F-Actor ?y]]

which forces BC on the Parent-of test for particular choices of F-Actor. Of course, this is a case of EQ that cannot be FC'd, but must be tested (it is not foldable) since the constrained variable is not ground at instance creation time. This is a somewhat obscure declaration, we can make it easier to express by using something like the POST function. e.g.

(6.25) `[POST [Parent-of [F-Object ?x] [F-Actor ?x]]]`

could be the semantic equivalent.
Another example: we may want to say that

```
(6.26)  [My-Predicate [F-Actor ?x]
                   [Any ?z [Another-Predicate [F-Object ?x] ?z]]]
```

Here is something else that is a perfectly valid *constraint*, but is not valid as an *assertion* (that is, the actor slot must make My-Predicate true for some other variable that some other predicate is true of). The whole idea here is that we may have *constraints* that are not *assertions*, and we should not confuse the two.

Define-Subtype also has a mechanism for setting up some standard relations. Once a relation has been defined using Define-REP-Relation *q.v.*, it can appear as a clause in the :Relations keyword option. Unlike Constraints and Initializations, Relations are not processed by the usual mechanisms, but left for the user to deal with via builtin functions. `[Relation-Form?]` and `[Relation-List]` are the builtins that can be used to manipulate the defined relations for a structured type. Note that relations are defined on a type, and not on an instance; it is up to the user to decide how to use the relations, *e.g.* bind ?self to a particular instance, prove them, *etc.*.

Here is an example. First we define a structured type whose subtypes will contain an assertable STEPS relation[4].

```
(6.27)  (Define-Subtype 'T-Plans-With-2-Steps 'T-U
             :Roles '((Agent T-Animate))
             ;; since we know we will have two steps...
             :Constraints '([EQ? [F-AGENT ?SELF]
                                 [F-AGENT [STEPS-1 ?SELF]]]
                            [EQ? [F-AGENT ?SELF]
                                 [F-AGENT [STEPS-2 ?SELF]]])
             :Initializations '([ASSERT-RELATIONS ?SELF :STEPS]))
```

What 6.27 does for us is define a type with one role (an Agent), and says that any time we instantiate some instance of the type, "assert" the forms on the :STEPS relation. Further, constrain the agent of each of the two steps to be the agent of this instance. (We actually didn't need to do this as we will shortly see). Now define some subtype:

```
(6.28)  (DEFINE-SUBTYPE 'T-HUNT 'T-Plans-With-2-Steps
             :Relations '((:STEPS [C-GO-TO-WOODS [F-AGENT ?SELF]]
                                  [C-GET-GUN [F-AGENT ?SELF]])))
```

This defines two (by default, uninterpreted) steps on the T-Hunt type, which are in fact, constructor functions. Since this is a subtype of T-Plans-With-2-Steps, we

[4]Note that we do not need to use Define-REP-Relation for a :STEPS since it is a predefined relation.

inherit it's initializations, so when we instantiate some instance of T-Hunt, we will have it's :Steps relations asserted. Note that since we are using constructor functions in the steps that each take one argument, the agent, and this agent is defined as the agent role of the instance, we don't really need the inherited constraints from the parent type, though they do point out the use of the constructed step accessors that Assert-Relations builds.

So, lets see what happens when we do (Define-Instance `[Hunt-1]` 'T-Hunt :Agent [Robert]):

1. We set the type of the function term `[Hunt-1]` to be T-Hunt.
2. Because T-Hunt is a structured type, we assert its role relations, namely `[Add-EQ [F-Agent Hunt-1] [Robert]]`.
3. We run the initializations on the instance. This binds ?self appropriately, so we try to prove `[Assert-Relations Hunt-1 :Steps]`. Since the type of `[Hunt-1]` is T-Hunt, we use the appropriate relations for that type. We thus assert `[Add-EQ [C-Go-To-Woods [F-Agent Hunt-1]] [Steps-1 Hunt1]]` and `[Add-EQ [C-Get-Gun [F-Agent Hunt-1]] [Steps-2 Hunt-1]]`. The constructor functions will also be appropriately instantiated (*e.g.* for the first one we will do something like `[Add-EQ [F-Agent [C-Go-To-Woods [F-Agent Hunt-1]]] [F-Agent Hunt-1]]`.
4. We attempt to prove or assert the constraints (that are inherited from T-Plans-With-2-Steps. Since by definition these are provably true, we do not need to add any additional information.

Note that Define-Subtype (as well as Define-Functional-Subtype, below), may only refer to functions that have already been typed, or refer to the function being typed, in the Initializations, Relations, or Constraints field. That is, one should not use any constructor functions, or regular types functions that one wants to use Declare-FN-Type on in these fields before these calls, because Rhet will incorrectly expand the type.

Define-Functional-Subtype *Type Supertype* &Key *(Roles* (<Rolename Type>)**) (Relations* (Relationname Relation-Forms*)**) Constraints Initializations*
This defines Type in the same manner as the Define-Subtype function, but in addition defines a constructor function for the type. Thus, given the definition of T-ACTION above,

(6.29) (define-functional-subtype 'T-EAT 'T-ACTION :roles '((R-OBJ T-FOOD)))

would define T-EAT to be a subtype of T-ACTION with roles R-OBJ and R-ACTOR (inherited), and would define the function f-obj for the R-OBJ role and a constructor function c-eat. This is roughly equivalent to adding:

(6.30) (Tsubtype 'T-ACTION 'T-EAT)

where the functions are defined by

(6.31) (Declare-FN-Type 'f-obj '(T-EAT) 'T-FOOD)

(6.32) (Declare-FN-Type 'c-eat '(T-FOOD T-ANIM) 'T-EAT).

although the c-eat function has the side effect of defining a new instance. The definition of f-actor for T-ACTION will apply as needed to instances of T-EAT.

Defining an instance of a functional subtype works just like defining non-functional subtypes, and the above example for T-Hunt would apply here as well if it were functional, with one exception: before we run the initializations we would establish a constructor function equivalent to [Hunt-1], and assert [Add-EQ [C-Hunt Robert] [Hunt-1]].

Note that **Define-Functional-Subtype** (as well as **Define-Subtype**, above), may only refer to functions that have already been typed, or refer to the function currently being typed, in the Initializations, Relations, or Constraints field.

Define-Conjunction *New-Type* (Existing-Type*) &Key *:Roles* ((New-Role-Name Existing-Role-Name*)*)
This defines New-Type to be a conjunction of the properties of the Existing-Types. Where one type may define the AGENT to be the ACTOR and another call it the AGENT, the :roles keyword allows explicitly informing RHET that these are indeed the same slot and which name should be used for access to it from the conjunction type.

Define-REP-Relation *Relation-Keyword* &Key (Inherit-type :Inherit) *FN-Definition-Hook*
Define Relation-Keyword (a keyword) to be an allowable relation for future calls on Define-Subtype and Define-Functional-Subtype. These are the currently defined Inherit types:

Mod

- **:Inherit** If this relation is given when defining a subtype, it is concatenated to the inherited definition.
- **:Redefine** The local relation definition takes priority over the inherited one (the inherited definition is only used if there is no local definition).
- **:Local** No relations are inherited at all from parent types.
- **:Merge** A special usage is handled, specifically, the relation entry is expected to be an Alist; children's entries are unified with their parent's and this becomes the child's 'actual' entry.

Note that REP-Relations must be defined before any use, and redefinition will only effect definition that occur afterwards.

Rhet predefines the relations :**Steps**, :**Preconditions**, and :**Effects** as a convenience to the user.

The FN-Definition-Hook will be called on the new type and list of relations for this relation type just before reparsing (typing) the constraints, relations, and initializations of a **Define-Subtype** or **Define-Functional-Subtype**. This allows the user to appropriately use **Declare-FN-Type** or **Add-FN-Type** for relation accessors she may be using. An example of this usage is given in [Miller, 1990a]. Writing this function is a good reason to use the :**Merge** inheritance type; the accessor for a particular entry will be conveniently part of the data, and the rules for constructing it need not be dealt with. New

The Rhet system provides a convenient abbreviated form for defining instances of structured types.

Define-Instance *Instance Type* &Rest <<*Rolename*> <*Value*>>* &Key *Context*
This defines Instance to be an **Itype** of Type and defines the indicated roles of Instance to have the indicated values. Instance and each Value are Rhet Forms. A context keyword and value may be supplied, but they must appear last in the argument list.

For example,

(6.33) (Define-Instance [E1] 'T-EAT 'R-OBJ [F1] 'R-ACTOR [JOE])

is equivalent to adding[5]

(6.34) (Itype 'T-EAT [E1])

(6.35) (Add-Role [E1] 'R-OBJ [F1])

(6.36) (Add-Role [E1] 'R-ACTOR [JOE])

with the main difference being that **Define-Instance** is a Lisp function (only), and the others are (potentially) horn clauses that could be added during a proof.

Either way of asserting this information will cause the following equalities to be derived:

(6.37) (Add-EQ `[c-eat F1 JOE] [E1]`)

(6.38) (Add-EQ `[f-obj E1] [F1]`)

(6.39) (Add-EQ `[f-actor E1] [JOE]`)

Further, note that the builtin `[Add-Role]`, is also available as a Lisp function:

Add-Role <*Instance*> &Rest <<*Role-Name*> <*Value*>>* &Key *Context*
Adds that object <Instance> has role <Role-Name> with value <value>. All arguments must be fully grounded. Note that Instance and each Value are expected to be Rhet Forms. Unlike the builtin, the lisp function will also take an optional Context keyword argument pair. If supplied, it must be last.

[5] Actually, these two calls to Add-Role could have been combined into one. Note that Add-Role is also available as a Rhet Builtin, as well.

6.3.2 Retrieving Structured Type Information

The Rhet system provides a general facility for providing information about any object defined. This is provided by the following functions:

Classify *Type :Roles* ((Role-Name Role-Type))*

This will return the (simple) types that are subsumed by Type and subsume types which have roles at least as general as the ones explicitly given. As an example, assume we have the following in our KB: The Action type has the role R-Agent which is required to be a Person; the ObjAction type takes the roles R-Agent restricted to Person, and R-Obj restricted to T-U, and the Drive type with roles R-agent restricted to be of type Person, and R-Obj restricted to be of type Automobile. In this case were we to do

(6.40) (Classify 'Action :Roles '((R-Agent Person)(R-Object PhysObj)))

we know the types subsumed by Action are ObjAction and Drive, but since ObjAction is more general than our query (specifically the R-Object role defined to be T-U which would subsume our own R-Object's type), we returns the type Drive since each role our query subsumes the defined roles of Drive.

Rep-Structures

Return a list of all structured types defined.

Retrieve-Def *Object*

which returns a description of the object as per the formats in table 6.2. Note that while the table shows a list being returned, Retrieve-Def actually returns multiple values.

For example, given the definition of T-OBJ-ACTION above, (Retrieve-Def 'T-OBJ-ACTION) would return:

(6.41) (TYPENAME (T-ACTION) (R-OBJ R-ACTOR) (T-PHYS-OBJ T-ANIM))

For another example, if A is an instance of T-OBJ OBJ-ACTION with the R-OBJ role set to O1 and R-ACTOR set to (f-actor A2), then (Retrieve-Def [A]) would return:

(6.42) (CONSTANT T-OBJ-ACTION (R-OBJ [O1]) (R-ACTOR [f-actor A2])).

Finally, given a function containing unbound variables, Retrieve-Def will return as much information as it can derive using the basic format for constants, but differing in the first atom, *i.e.*, it returns

(6.43) (FUNCTION <type> (<role> <value>)*)

Type of Object	Return Values:	(actually uses multiple values)
Type	(:TYPE-NAME	(<immediate supertypes>) (<roles defined>) (<role type restrict.>) (<foldable non-monotonic constraints>) (<foldable monotonic constraints>) (<non-foldable non-monotonic constraints>) (<non-foldable monotonic constraints>) (<initializations>) ((<relation-name> <relation-form>*)*))
Rolename	(:ROLE-NAME	<list of types using that role>)
Function Name	(:FUNCTION-NAME	(<type restrictions on args> <type of result>)*)
Relation	(:RELATION-NAME	(<types using that relation>))
Free Variable	(:VARIABLE	<type restriction>)
Constant	(:CONSTANT	<Type> (<role> <value>)*)
Func. w/ unbound vars	(:FUNCTION	<type> (<role> <value>)*)
Anything Else	(:UNKNOWN	)

Table 6.2: Object Descriptions Returned by Retrieve-Def

as mentioned in table 6.2.

For example, given the assertions of 6.18, 6.29, and 6.33, we would retrieve the following:

```
Rhet-> (retrieve-def 'T-EAT)
(:TYPENAME (TYPEKB::T-ACTION) (R-ACTOR R-OBJ) (T-ANIM T-FOOD) NIL NIL
NIL NIL NIL NIL)
Rhet-> (retrieve-def 'R-OBJ)
(:ROLE-NAME (TYPEKB::T-EAT))
Rhet-> (retrieve-def 'R-ACTOR)
(:ROLE-NAME (TYPEKB::T-EAT TYPEKB::T-ACTION))
Rhet-> (retrieve-def 'f-obj)
(:FUNCTION-NAME (((T-ANYTHING) . *T-ANYTHING) ((*T-EAT) . *T-FOOD)))
Rhet-> (retrieve-def 'c-eat)
(:FUNCTION-NAME (((*T-ANIM *T-FOOD) . T-EAT)))
Rhet-> (retrieve-def ?x*T-EAT)
(:VARIABLE T-EAT)
Rhet-> (retrieve-def [E1])
(:CONSTANT T-EAT (R-ACTOR [JOE] R-OBJ [F1]))
Rhet-> (retrieve-def [f-obj E1])
(:CONSTANT T-FOOD NIL)
Rhet-> (retrieve-def '[c-eat JACK [f-obj E1] ])
(:Function T-EAT (:Dont-calc-roles-for-constructors-yet))
Rhet-> (retrieve-def '[c-eat JACK ?x*FOOD ])
(FUNCTION T-EAT (:Dont-calc-roles-for-constructors-yet))
Rhet-> (define-instance [E2] 'T-EAT 'R-ACTOR [JACK])
[E2]
Rhet-> (retrieve-def [E2])
(:CONSTANT T-EAT (R-ACTOR [JACK] R-OBJ [f-obj E2]))
Rhet->
```

Subsumes *Type* (Role-Name Role-Type)*

Not in V17.9

This succeeds if the given roles and value restrictions all subsume those of Type.

In general, an unknown type is considered to subsume a known type if:

1. The preliminary guess as to the type of the unknown subsumes the type of the known.
2. The roles of the known are a subset of those of the unknown.
3. Each of the role restrictions on each of the roles of the known type are subtypes of those of the unknown.

6.3.2.1 Dealing with Relations

New

Some special functions are defined to assist in manipulation of type relations by lisp programs. The contents of the relations field is made reasonably generic in order to facilitate general use, however, many more functions (from [Miller, 1990b]) may also well prove useful.

Get-Relations *Relation Type*
: retrieves the relations of type Relation from the type structure Type. Type may be either a symbol representing a type, or an **Itype-Structure**; the latter can be generated either by the #! reader macro, or by the **Make-I-Type** function.

With-Self-Bound *(Form Binding* &Optional *Leave-Bound Error-String)* &Body *Body*
: All ?self variables in Form are bound to Binding within the scope of this macro, or permanently if the Leave-Bound flag is non-nil. If the variable cannot be bound, an error is generated, and Error-String is appended to the message "I could not unify ?self*T-self-type to binding".

Chapter 7

Debugging Tools

New

Rhet currently supports four sorts of debugging tools: the first is the ability to turn on a trace of specific axioms, or equalities, or the usages of a particular term. This is documented in the next section, 7.1. The second would be the various hook functions provided to intercept multiple proofs, which is documented in section 7.2. Third are the various low-level tracing facilities normally used to debug Rhet itself, but possibly useful for pernicious problems. These are documented in section 7.3. And last, in section 7.4, we outline a few strategies for using some normal builtin functions to help debug code.

7.1 Higher Level Tracing

The following functions are provided for the user to request tracing. Note that all have equivalent command-level and menu operations described in section 8.2.3, which may be easier to use, since these functions generally require the actual interned term to be passed in order to establish tracing.

Trace-BC-Axiom &Optional <*BC-Axiom or Form*> (How '((:Call . :Trace) (:Return . :Trace)))

With no arguments, returns a list of BC axioms traced, otherwise begins tracing the passed axiom. If a form is passed instead of an axiom, all BC-Axioms whose LHS unify with the passed form are traced. How should be an alist whose car is:

:CALL trace entry

:NEXT trace calls to RHS terms

:RETURN trace proof/failure return

:RETRY trace failures from RHS terms

and whose cdr is the Action which should be one of:

:TRACE just print a diagnostic

:BREAK invoke a break loop

:DEBUG invoke the debugger

:STEP invoke the stepper

The BC-Axiom passed should be the actual axiom, *e.g.*as returned from List-B-Axioms, if a Form is not supplied instead.

Trace-EQ-Object &Optional *Function-Term* (How '((:Eq . :Trace) (:Ineq . :Trace)))
With no arguments, this returns a list of equality objects traced, otherwise begins tracing the passed Function Term object. How should be an alist whose car consists of:

:EQ trace changes to this term's equivalence class

:INEQ trace changes to this term's inequivalence class

:CLOSURE trace changes to any term that references this one.

and whose cdr is the Action which should be one of:

:TRACE just print a diagnostic

:BREAK invoke a break loop

:DEBUG invoke the debugger

For example,

(7.1) (Trace-EQ-Object [A])

would cause a diagnostic message to be printed any time [A]'s equivalence or inequivalence class changed.

Trace-FC-Axiom &Optional < *FC-Axiom or Form* > (How '((:Call . :Trace) (:Return . :Trace)))
With no arguments, returns a list of FC axioms traced, otherwise it begins tracing the passed FC axiom. If a form is passed instead of a specific FC axiom, all FC axioms whose trigger unifies with the passed Form are traced. How should be an alist whose car is:

:CALL trace entry (trigger)

:NEXT trace calls to RHS terms

:RETURN trace proof/failure return

:RETRY trace failures from RHS terms

and whose cdr is the Action and should be one of:

:TRACE just print a diagnostic

:BREAK invoke a break loop

:DEBUG invoke the debugger

:STEP invoke the stepper

The FC-Axiom passed should be the actual axiom, *e.g.*as returned from **List-F-Axioms**, if a Form is not passed instead.

Trace-Request &Rest *Function-Terms*
Request a trace of anything involving any of the passed Function Terms.

Untrace-BC-Axiom &Optional < *BC-Axiom or Form* >
With no arguments, untraces all BC axioms. Otherwise untraces the passed BC axiom, or all traced BC axioms whose LHS unify with the passed Form.

Untrace-EQ-Object &Optional *Function-Term*
With no arguments, untraces all equality objects. Otherwise untraces the passed Function Term.

Untrace-FC-Axiom &Optional < *FC-Axiom or Form* >
With no arguments, untraces all FC axioms. Otherwise untraces the passed FC axiom, or all traced FC axioms whose trigger unify with the passed Form.

Untrace-Request &Rest *Function-Terms*
request a trace of anything involving any of the passed Function Terms.

7.2 Hook Functions Related to Debugging

PROOF-TRACE This variable is set to the justifications for the proof of the last full goal (using proof-entry structures).

REASONER-PAUSE-FUNCTION If bound to some function, the function will be called between proofs during a **Prove-All**, **Prove-Complete**, *etc.* If it returns **Nil**, a Lisp abort is signalled. This is to allow the user interface to support doing only a limited number of proofs, or querying the user between proofs to see if he wants to continue. It is called with the forms already proven (with bindings).

REASONER-STEP-FUNCTION If bound to a function, it is called between proof macro steps. If it returns **Nil**, the proof is aborted. This is to allow the user interface to insert a break after so many steps of the Reasoner, as it did in HORNE [Allen and Miller, 1986]. It is called with the form to be proved (with current bindings) at the current state of the system as it's only argument.

TRACE-FORWARD-ASSERT If non-nil, each chained assertion is printed on the error output.

TRACE-REASONER If non-nil, a diagnostic for each proof decision is printed on the error output. If it's value is **:BUILTIN**, builtins are traced as well.

PROOF-DEFAULTS-USED This is set to the list of default facts that were used or gotten via a default axiom in the current proof. It will be a list of lists if a **Prove-All** was used.

7.3 Lower Level Tracing

Most of the lower level tracing can be effected by the :Set Rhet Process Status command in the high level user interface. The variables that this interface changes are described here:

OMIT-OCCURRENCE-CHECK Normally non-nil, which keeps the unifier from doing an occurrence check. Can be set to nil to help track down recursive unification bugs.

REASONER:*REASONER-DISABLE-EQUALITY* Normally nil, setting non-nil turns off equality.

REASONER:*REASONER-DISABLE-TYPECHECKING* Normally nil, setting non-nil turns off typechecking.

REASONER:*REASONER-ENABLE-DEFAULT-REASONING* Normally non-nil, setting nil turns off default rule access.

REASONER:*TRACE-ASSERT* Setting non-nil will trace all (non-equality) assertions.

REASONER:*TRACE-FORWARD-ASSERT* Setting non-nil will trace all FC actions.

REASONER:*TRACE-REASONER* Setting non-nil will trace all BC actions. Setting to :Builtin will also cause tracing of builtin functions, which is otherwise suppressed.

E-UNIFY:*TRACE-HEQ* Setting non-nil will display all equalities added.

E-UNIFY:*ENABLE-HEQ-WARNINGS* Setting non-nil will allow some generally uninteresting warnings to print out.

RHET-TERMS:*HDEBUG* Setting non-nil will not only allow some generally uninteresting warnings to print out, but also disable some sanity checks; this is used internally when the caller knows what it's doing, and it's OK to delete, for instance, the ROOT context.

7.4 Strategies for Debugging

Rhet does not currently include a PROLOG-style four port debugger. Partially, this is because RHET does not define an evaluation order, and further the possibility of intelligent backtracking complicates things. The easiest first approach to debug is simply to turn on tracing for predicates you care about, and iteratively deepen the number of predicates you trace (e.g. if you don't get the response you expect, turn on tracing for some of the predicates in the RHS of the form). If things become really bad, you could insert a [CALL `(progn (break) t)]` goal in the RHS; when Rhet goes into a break, you could then turn on some lower level tracing (which would otherwise swamp you with output if you just had it on all the time). In particular, it may be useful to know that for all the high level commands documented, you can access them from a lisp breakpoint by calling the function as RWIN:COM-<name-using-hyphens-instead-of-spaces>. So, for instance, to get the menu that :Set Rhet Process Status provides (for playing with debug variables, I could evaluate (RWIN:Com-Set-Rhet-Process-Status) instead; the former only working from a Rhet command prompt, and not a breakpoint or from within the debugger.

Chapter 8

Enhanced Interactive Interface

The basic idea of the enhanced interactive interface, is to enhance existing Lisp machine tools, or provide equivalents that are similar enough to the existing tools that they require no additional learning process. Therefore, the fundamental Rhet construction tool is be ZMACS, and the fundamental Rhet interaction mode is a Rhetorical System Window. Tracing and debugging should be orthogonal to normal Lisp tracing and debugging on the Lisp machine, *e.g.* from ZMACS, m-x Trace should prompt for the function to trace (and default to the function the cursor is on) and then a pop-up menu will present the trace options in the same manner the normal Lisp trace function would[1].

8.1 Editing Rhet Code

Rhet code should be created and edited in a ZMACS buffer. To support this, there is a new lisp syntax called Rhet, since it uses the readtable that understands Rhet clauses. The user's Lisp code that may interact with the Rhet system could also appear in such files. The Rhet syntax merely binds the readtable so things like square brackets are correctly interpreted as Rhet terms.

Rhet clauses that are added to the system via **Assert-Axioms** cannot be handled in terms of, *e.g.*, redefinition. The user is instead advised to use **DefRhetPred**, which will correctly handle re-evaluation as a redefinition of the entire predicate group the **DefRhetPred** covers.

Rhet mode should (eventually) support the following:

- All the normal formatting, matching, *etc.* of Lisp mode.
- c-sh-e to add or redefine a Rhet axiom, inside a **DefRhetPred**.

Currently putting justifications on axioms in a file is not addressed.

[1] This goal, alas, has not yet been achieved, however.

8.2 Interacting with the Rhet System

There are two substantially different implementations of the Rhet window interface, one which runs on the Symbolics and using dynamic windows, and one which runs on the Explorer and confines itself to scrolling windows. They are similar in functionality in terms of available commands, however, the window history, top level commands, general mouse sensitivity, and simple incremental command addition the Symbolics provides is not possible with the current Explorer release. Only the Symbolics interface is documented here; a CLIM version should be generated in the next 12 months that will be portable between the lispmachine and other architectures. The Explorer version currently works but is unsupported.

The Rhetorical System Window Interface is a window oriented program designed for interactive use of the Rhet System. It is meant to be flexible and easy to use. The underlying structure is a frame made up of the following panes: a title pane, a Lisp pane, an item pane, and a main command menu pane. These can be configured in various ways. There are many parameters that the user can set himself either during a rhet session, or in his/her rhet initialization file. Furthermore all settable parameters have default values they assume if no others are specified. The frame is selectable via <Select>-R.

8.2.1 The Panes

The Title Pane This is nothing more than what the name suggests. This pane is always present at the top of the screen. It normally says "RHET Vers. 17.9(released)", along with an indication of optional subsystems loaded (*e.g.* RPRS or TEMPOS).

The Main Command Menu Pane This is a menu pane. It is always present and appears immediately under the title pane. These are abbreviations for Command Processor Commands, which immediately appear as top-level input on selection.

The Lisp Pane This is a Lisp listener window. The value of *terminal-io* is the Lisp pane. Therefore it is present in each configuration. Lisp commands can be typed in anytime and will be evaluated in this window.

The Item Pane This is a mouse-sensitive text-scroll-window. Its main use is for examining the Rhet knowledge base. Rhet objects that are placed in the item pane usually have a box appear around them when the mouse is positioned on them. Clicking on an object will cause a menu of operations to be performed on that object to appear. More information can be found in the programmer's guide [Miller, 1990b].

8.2.2 Pane Configurations

All configurations have a title pane, a main command pane, a and a Lisp pane.

Lisp This has one large Lisp pane and is the default configuration.

Vertical Display This has a vertical Lisp pane next to a vertical item pane.

Horizontal Display This has a horizontal Lisp pane on top off a horizontal item pane. This is the default configuration with an item pane.

Oversize Display This has a large item pane on top off a small horizontal Lisp pane.

Expert Lisp This is like the default configuration, but has no menu.

Expert Display This is like the Horizontal Display, but without a menu.

Expert Oversize Display This is like the Oversize Display, but without a menu.

8.2.3 Available commands

The Symbolics implementation takes advantage of the more advanced window facilities available under Genera.™ Items are displayed via their presentations, allowing them to be moused and reused at any time. Further, certain commands will be available on highlighted presentations via a left, middle or right click on the mouse. All commands are available as command input to the command processor, although a fair sample is *additionally* available via a command menu pane. A brief summary of the commands are listed here. Unless otherwise specified, they are all equivalent to calling the lisp function of the same (or virtually identical) name, though the arguments will be prompted for and often allow completion on input and other facilities. Output of some commands is graphical in nature as well, to enhance the density of the information displayed.

Note, however, that in addition to the commands listed here, some existing Genera commands have been modified to allow Rhet to better integrate with the environment. In particular, there is a Zmacs minor mode for dealing with Rhet files (which set the readtable appropriately), which is accessible via the **M-x Set Lisp Syntax** command from Zmacs, or :**Set Lisp Syntax** from a lisp listener. In addition to offering the Common-Lisp or Zetalisp syntaxes, this command will also offer the Rhet syntax. Further note that several presentation translators have been defined for rhet objects, so they will be both mouse-sensitive, and have appropriate commands for manipulating the objects on the left or menu (right) mouse keys.

:Add Eq This is also available from the command menu. It takes two forms, and choice of "Equality" or "Inequality" to make an appropriate assertion to the KB. A keyword option allows explicitly setting the context for the assertion.

:Assert This is also available from the command menu. It takes one axiom or fact as an argument, and asserts it to the KB. A keyword option allows explicitly setting the context for the assertion.

:Create Context This is also available from the command menu. It creates a new user context, prompting for it's name, and for the parents name. The parent context must already exist. See 5.4 for more information on user contexts (Ucontexts).

:Create Instance An interface to Itype, Dtype and Utype. This is also available from the command menu. It takes a type name for the type of the instance, then a sequence of forms which are to the the instances. Last, it prompts for a specific choice, either "Immediate", "Unspecific" or "Distinguished". See the definitions for Itype, Dtype and Utype for more information, in section 2.3. New Optionally, the term can be declared a Candidate Primary; see the descriptions of the above functions for details.

:Declare Disjoint This is also available from the command menu. It takes a sequence of types which are all to be declared disjoint.

:Declare Function Type This is also available from the command menu. It takes an atom as the function name which will be typed by this command, and a list of the type constraints. See the lispfunction Declare-FN-Type for more information, in section 6.1.

:Declare Intersection This is also available from the command menu. It takes a type name to be the name of the intersection, then a sequence of types to be intersected.

:Declare Overlap This is also available from the command menu. It takes a sequence of types to be declared to overlap.

New **:Declare Set Type** An interface to Declare-Set-Type, this command takes a new set type, an existing parent set type (must be a subtype of T-SET), and an element type. This will allow rhet to appropriately set the type of a set whose elements are all of the element type (or subtypes of them). Note that declarations here, as for :Declare Function Type, must be internally consistent. That is, a set type that is a subtype of some other set type must have an element type that is a subtype of the other set type's element type, and they MAY NOT BE EQUAL (otherwise Rhet could not classify them).

:Declare Subtype This is also available from the command menu. It takes a type as the parent or supertype, then a sequence of types which are all to be declared as subtypes of the parent. It is not necessary that the parent be already declared.

:Declare Partition This is also available from the command menu. It takes a parent type which will is the type that will be partitioned, then a sequence of child types, which all are subtypes of the parent as well as completely covering the parent and being themselves disjoint.

New **:Delete BC Axiom** This is also available from the command menu. It will put up a subsidiary menu that will allow the user to:

Remove Specific BC Axioms It prompts for a Form, and axioms whose LHS unify with the form are removed.

Remove Specific BC Axioms by Index It prompts for an index, and all axioms with that index are removed. Note that the index may be a regular expression, in which case all axioms whose indexes match the regular expression are removed. See 5.1.1.1, for more information on acceptable regular expressions.

Retract all Backward This will delete all BC axioms.

:Delete Contexts This is also available from the command menu. After asking for confirmation, all user contexts are deleted from the system.

:Delete Facts This is also available from the command menu.

:Delete FC Axiom This is also available from the command menu. It will put up a subsidiary menu that will allow the user to: New

Remove Specific FC Axioms Like it's BC counterpart, it prompts for a form, and axioms whose trigger unify with the form are retracted.

Remove Specific FC Axioms by Index Again, similar to the BC counterpart, but it retracts only FC axioms.

Retract All Forward Deletes all FC axioms.

:Delete Types This is also available from the command menu. After asking for confirmation, all type declarations (except for the predefined ones) are cleared from the type table.

:Expression Reader This is also available from the command menu. It reads expressions one at a time from the prompted for file. Useful for demonstrations and testing. The expression reader will echo comment lines (lines whose first non-blank character is ';') to the output as they are encountered.

:Find Fact This is also available from the command menu. It prompts for a form and returns a list of all Facts that unify with the form. A keyword option is available to specify a specific context, otherwise the default context is used.

:Find Facts By Index This is also available from the command menu. This command prompts for an index, and returns each fact in the KB with an index that matches it. The index is assumed to be a regular expression, (See 5.1.1.1, for more information on acceptable regular expressions) unless the keyword option ":exactly" is supplied. Again, a keyword context option is allowed.

:Find Fact References This command prompts for a Form which can be coerced into function terms, and it displays the set of all facts that reference these function terms (*i.e.* have them on their arglist).

:Find Facts With Bindings This is also available from the command menu. Similar to :Find Fact, but it also returns the bindings needed to unify the form prompted for with each fact found in the KB. Again, a keyword context option is allowed.

:Prove This is also available from the command menu. It takes a sequence of legal goals, and then attempts to prove them using the BC mechanisms. A keyword option, :ALL, may be supplied, which will instruct Rhet to return all possible distinct proofs. The keyword option, :HOW, allows specification of the proof mode, either Simple, Question-Answering (the default), or Complete. Another keyword option allows explicitly setting the context for the proof.

:Reset Rhetorical This is also available from the command menu. After asking for confirmation, the rhetorical system is reset to the state it would be in immediately after loading it. Note that if Rhet was saved in a world with some user state already defined, this will be cleared as well. New In addition, this command will allow Rhet to be reset without resetting the proof modes and options.

:Set Rhet Window Configuration This is also available from the command menu as "Configuration". This allows the user to alter the gross ratio of the size of the Lisp Listener to the size of the Item Pane, and optionally delete the menu pane.

Mod **:Set Rhet Process Status** This is also available from the command menu as "Status". This displays a list of configurable variables in menu form, and accepts changes from the user.

:Show BC Axiom This is also available from the command menu. This prompts for an atom, the predicate the user is interested in. All BC axioms that are defined on that predicate head are displayed. A keyword option is available to specify a specific context, otherwise the default context is used.

:Show BC Axioms By Index This is also available from the command menu. This prompts for an index, which may be a regular expression, and displays all BC axioms with that index. A keyword option, "Exactly" is available, which forces an exact match with the index rather than to take the index as a regular expression. (This is faster if the index doesn't have any regular expression operators anyway.) See 5.1.1.1, for more information on acceptable regular expressions. A keyword option is available to specify a specific context, otherwise the default context is used.

:Show Contexts This is also available from the command menu. Displays the UContexts, as a graph, in a separate window. A keyword option, ":internal" is available to show the internal Rhet contexts instead of the Ucontexts. See 5.4 for more information on user contexts (Ucontexts).

:Show Equivclass This is also available from the command menu. It prompts for a function term, and displays a list of all other function terms that are equivalent to it. A keyword option is available to specify a specific context, otherwise the default context is used.

:Show FC Axiom This is also available from the command menu. It acts like :Show BC Axiom, except on FC axioms. A keyword option is available to specify a specific context, otherwise the default context is used.

:Show FC Axioms By Index This is also available from the command menu. It acts like :Show BC Axioms by Index, except on FC axioms. A keyword option is available to specify a specific context, otherwise the default context is used.

:Show File Like the normal Genera command, but views the document in the viewer pane.

:Show Form Equivclass This command prompts for a form, and displays all function terms that could unify with the form, or are equivalent to function terms that could unify with the form. A keyword option is available to specify a specific context, otherwise the default context is used.

:Show Function Type This command prompts for an atom (the function), and describes the function typing declarations that have been made on the function.

:Show Inequivclass This is also available from the command menu. It prompts for a function term, and displays all other function terms that are in the closure of those that have been declared to be inequivalent to it. A keyword option is available to specify a specific context, otherwise the default context is used.

:Show Rhetorical Help This shows a very introductory help file on the window related facilities. It is available from the command menu as "Help".

:Show Set Types Similar to :Show Function Type, it displays a list of all the typing information for classifying sets that have been declared to rhet either via :Declare Set Type or Declare-Set-Type. New

:Show Structured Type Info This prompts for a structured type name (*i.e.* one defined with either Declare-Subtype or Declare-Functional-Subtype). It displays the various slots, initializations, and constraints as defined or inherited for this type.

:Show Type Info This prompts for a type name, and displays its relationship, if known, between itself and all other declared types.

:Show Typed Functions This is also available from the command menu. It lists all functions that have been defined to have a type.

:Show Types This is also available from the command menu. Displays all the types Rhet knows about, as a graph showing subtypes.

:Start Dribbling This is also available from the command menu. Turns on the session scripter. See section 9.1 for more information[2].

:Stop Dribbling This is also available from the command menu. It turns off the session scripter.

New

:Trace BC Axiom Clicking or typing a BC axiom will allow it to have various diagnostics printed. For more information on the options, examine the Trace-BC-Axiom and Untrace-BC-Axiom functions, which this command is an interface to. As with the underlying function, a Form is also a valid argument, in which case any BC axiom whose LHS unifies with the Form are traced (or untraced).

:Trace EQ Object Clicking or typing a function term will cause a diagnostic to be printed whenever it is involved in certain types of equalities. For more information on the options, examine the Trace-EQ-Object and Untrace-EQ-Object functions, which this command is an interface to.

:Trace FC Axiom Clicking or typing an FC axiom will allow it to have various diagnostics printed. For more information on the options, examine the Trace-FC-Axiom and Untrace-FC-Axiom functions, which this command is an interface to. Similar to these underlying functions, a Form is also a valid argument, in which case any FC axiom whose trigger unifies with the Form are traced (or untraced).

:Trace Term This is similar to :Trace EQ Object, but only works on function terms, and causes a diagnostic to be printed whenever any term that references the passed term is touched. See also the Trace-Request and Untrace-Request functions, which this command is an interface to.

[2]Note that the session scripter is not just like a dribbling lisp listener!

Chapter 9

Options

This section outlines the various flags and options the user can set, typically before starting a proof.

9.1 Interaction Log

Since the normal mode of operation for large Rhet systems will be to save a world with all the axioms and facts added that are necessary, Rhet supplies functionality to allow the user to capture a log of what he has done during a session[1]. This is captured as a Lisp file: the user will be able to load the file and "replay" Rhet on his previous session if need be. Note that this is not the recommended way to save axioms: it will be much slower than just having used the axiom compiler and loading the compiled axioms directly[2]. The intent would be that proofs, and other calls could be captured, along with Rhet's output (put into the file as comments), which will allow a user to debug her interface much quicker than if he must do everything interactively.

The user enables the logger similarly to a dribbling Lisp-listener. Evaluating **(Rhet-Dribble-Start file-spec)** will start the logger, and **(Rhet-Dribble-End)** will finish logging and properly close the file. Note that changes made via the editor are not captured, only the online session with Rhet will be caught in the file.

For reference:

Rhet-Dribble-Start *File-Spec* &Optional (Mode :Both)
: Enable recording of Rhet interactions into the file. The default mode is :**Both**, that is both queries and assertions will be recorded, otherwise use :**Query** or :**Assert** to be selective. In either case, the functions are represented in the file, along with their results commented out. You can change the mode while dribbling by supplying a Nil File-Spec. The File-Spec may also be a stream. In this case, the function will not attempt to open the stream, just use it.

[1] This is available from the Rhet Window as the File Dribbler.

[2] which is much slower than loading a world with the axioms already loaded.

Rhet-Dribble-End
Disable recording of Rhet interactions.

9.2 Type Assumption Mode

The default mode for Rhet is to assume that two types whose relationship is not known are not compatible. This can be overridden by evaluating (Set-Type-Mode :ASSUMPTION), in which case, all unknown relationships are assumed to be unnamed intersections. Alternatively, setting the mode via (Set-Type-Mode :QUERY) will query the user each time two types are found for which there is no known relationship. Evaluating (Set-Type-Mode :DEFAULT) returns the system to default mode.

In assumption mode, the format of answers is unchanged, but the prover will return an additional value: that of type assumptions. Thus the first value will be the answer, the second will be the bindings, and the third will be the type assumptions used.

For example, given `[Q ?x*CAT]` and proving `[Q ?x*DOG]` in assumption mode where no relationship is known between the types *CAT and *DOG, we get three values:

```
[Q ?x*(CAT DOG)]
(?x ?x*(CAT DOG))
((CAT DOG))
```

Note that if you obtain multiple answers in this mode, the list of assumptions for each answer may refer to assumptions needed for other answers as well.

9.3 Constraint Assumption Mode

The default mode[3] for Rhet is to only allow two constrained variables to unify if there is at least one instance of the new restricted variable. For example, if we wish to unify:

(9.1) `[any ?x [PA ?x]], [any ?y [PB ?y]]`

we only allow the unification to proceed to

(9.2) `[any ?z [PA ?z] [PB ?z]]`

if there is some object ?z such that `[PA ?z]` and `[PB ?z]`. We do not bind the variable to it, but this is used as a quick check that constrained variables do not get out of hand. By calling the function (Set-Constraint-Mode :ASSUME), this check is turned off. If two constrained variables would have unified ignoring the constraints (*i.e.* their types are compatible) they are unified. (Set-Constraint-Mode :QUERY) will cause Rhet to ask the user every time a constrained variable is to be unified if it should proceed, and (Set-Constraint-Mode :DEFAULT) to return Rhet to its normal constraint processing.

No additional output is produced.

[3]Current Status: always :ASSUME

9.4 Equality Assumption Mode

The default for Rhet[4] is only to allow two constants to unify if they are identical, or provably equal. (Set-Equality-Mode :ASSUME) will turn on a mode that will cause Rhet to allow any two constants to unify, unless they are provably disjoint. As usual, (Set-Equality-Mode :QUERY) and (Set-Equality-Mode :DEFAULT) do the same things as the above. Note that the user will not be queried about constants that can be proved to be equal.

The proof returned will include an extra value: a list of constants that have been unified:

<answer> <variable bindings> <equality assumptions>

If both equality and type assumption mode are engaged, the type assumptions will appear first.

9.5 Reasoning Mode

By default, Rhet uses Default Reasoning Mode, as described in the tutorial. Complete Reasoning Mode can be set by calling (Set-Reasoning-Mode :COMPLETE); Simple Reasoning Mode can be set by calling (Set-Reasoning-Mode :SIMPLE) and finally Default Reasoning Mode can be returned to by calling (Set-Reasoning-Mode :DEFAULT).

9.6 Default Context

The system starts up with *Default-Context* set to the result of

(9.3) (Operator "MBSB" (Convert-Name-to-UContext "T")).

The user may set this to the result of the UContext or Operator functions. This is used for the context axioms and facts are asserted to if the context isn't specified with the call, and for the context to do proofs in, again, if not specified.

9.7 Contradiction Handling Options

Contradictions will always be reported (and put into the log, as per section 9.1), but the user can decide among the following options how he wants contradictions to be handled:

- By default, contradictions are ignored, and the tree is pruned in complete reasoning mode, unless it is the top-level goal that is contradictory, in which case the proof will fail, returning :Contradiction. This mode can be enabled via (Set-Contradiction-Mode :DEFAULT).
- The user can have the Rhet debugger invoked as soon as a contradiction is found. This is set by evaluating (Set-Contradiction-Mode :DEBUG).

[4]Current Status: always :DEFAULT

- The user can also supply his own function for handling contradictions. In this case, when a contradiction has been detected, a value will be **Thrown** to the user. Thus he should wrap his call to the prover with an appropriate **Catch** form, and call **(Set-Contradiction-Mode** :THROW **label)** where label is the name of his catch form (the label that will be thrown).

9.8 Enhanced User Interface Parameters

9.8.1 Example Initialization File

When Rhet is first started (it's window is first selected), like other applications on the lisp machine, it will load an init file: "rhet-init.lisp" from the user's home directory.

```
;;; -*- Mode: Lisp; Syntax: Rhet; Package: RHET-USER; -*-

;;; put any forms here I want executed when I log in,
;;; first use Rhet, and are not dependent on what
;;; system I am running.

(setq *reasoner-pause-function* #'my-pause-function)
(cond ((and (boundp 'dw:*program*) ; on the symbolics
            ;; currently in rhet frame?
            (typep dw:*program* rwin::rhet))
   (setf (send dw:*program* :reasoner-pause-function)
         #'my-pause-function)))
```

Appendix A

Version 17.9 Notes:

A.1 Changes since last major release

The following significant changes (not bugfixes) to Rhet functionality and user interface have been made since the last release of this manual (15.25):

New

15.27 fully implemented Set-Contradiction-Mode.

15.28 fully implemented Get-Type-Object.

15.31 defined another relation inherit type. We had the ability to have a child just redefine a relation (:Local), inherit the parent's relations (:Inherit) concatenating anything mentioned locally, or redefine only the ones it mentions (:Redefine). Now a new relation inheritance type is defined: :Merge. This is similar to :Inherit, but the relation list is expected to be an alist. If the child defines a relation whose key matches the an entry in the parent, the child's has priority and replaces it.

15.32 updated the way type constraints are handled; setting a var's type s.t. a constraint is satisfied will cause removal of the constraint. (It would have succeeded anyway, but this keeps the constraint list less cluttered and thus easier to debug).

15.35 modified the expression reader to no longer preparse files, and echo comment lines to the output.

16.0 did (at least) the following:

1. Reorganized the code somewhat (packages TYPE and UIC were combined into RHET-TERMS, and HEQ and UNIFY were combined into E-UNIFY; mainly to decrease cross-package references.
2. Added constructor-set to the canonical structure, and then modified equality and canonical name handling to correctly handle constructor functions at this low level. This also involved FLAG annotations on function-terms and forms.

(Prior to release 16, making [C-FOO A B C] eq to [C-FOO D E F] incorrectly did not also assert that A and D were now eq, etc. (remember functional subtypes declare that instances are unique, so this equality is an expected inference). It was never handled in HORNE, and we "knew" it had to be handled "manually" but now Rhet does it right.

3. To enforce the correctness of the above, Define-Subtype and Define-Functional-Subtype now enforce the following rules:

   - rolenames must be of the form R-RoleName.
   - types must be of the form T-TypeName.
   - slot-accessor functions must be of the form F-RoleName.

   Violations of the first two will signal an error and the type will not be defined. Violation of the last will prevent recognition of the term as a slot-accessor function.

16.3 Add the variable *Print-FN-Term-Pretty* to control pretty printing of function terms. If non-nil (the default) the primary instance of a canonical class will be printed for any function term, if false, the exact function term will be printed.

16.4 Enhanced Define-Functional-Subtype to allow multiple functional definitions. Previously, no subtype of a functional subtype could be functional.

16.6 Add handling of non-monotonic constraints.

16.8 A number of hook variables were added to allow easy integration with user driver code. More information is available in [Miller, 1990b].

16.14 New macros and functions to facilitate a user dealing with relations: With-Self-Bound and Get-Relations.

16.22 added a new trace variable, *Trace-HEQ*, which prints a diagnostic on each equality added. (Normally, the :Set Rhet Process Status command would be used to update this variable).

16.24 Relation hooks were added; see Define-REP-Relation's FN-Definition-Hook argument for more details. Define-Conjunction finally became fully operational.

16.27 DefRhetPred enhanced to allow a null body; the head would be used as a declaration form.

16.32 Add new Type-Relation builtin.

16.37 Added hooks for user lisp code to handle Rhet errors encountered. This is further described in [Miller, 1990b].

16.38 When Declare-FN-Type or Add-FN-Type is used, previously constrained variables are checked to see if they can be simplified with the new information.

16.42 added a new trace variable, *Trace-Assert*, which prints a diagnostic on each non-equality assertion added. (Normally, the :Set Rhet Process Status command would be used to update this variable).

16.48 FC enhanced to undo any assertions chained if a contradiction is found before the net settles.

16.51 Assert enhanced to undo partials if an error is encountered. Thus Assert-Axioms, and the builtins [Assert-Axioms] and [Assert-Fact] are now, basically, atomic operations.

16.52 Add an option to Reset-Rhetorical to allow modes to not be reset. Add new axiom tracing commands: Trace-BC-Axiom, Untrace-BC-Axiom, Trace-FC-Axiom, Untrace-FC-Axiom, Trace-EQ-Object, Untrace-EQ-Object, and deleted references to old unimplemented tracing stuff in the user's menu. All of the above also have command forms. Add a new builtin: [Expand-Constructor] to force expansion of a constructor function if necessary.

16.55 Add :Candidate-Primary keyword option to Itype, Dtype, and Utype, as well as :Create Instance.

16.56 Add :Trace Term command and Trace-Request and Untrace-Request functions. These allow tracing to occur when a term is referenced, rather than used directly.

17.0 Added [Skolemize] builtin; added don't-care variables, added new predefined types for axioms, facts, types, and numbers, added support for sets (builtins [Set-Setvalue] and [Fix-Cardinality]), added support for numbers (builtins [:=], [==], [=/=], [>], [<], [>=], and [<=]).

17.6 Changed the Ground builtin to succeed if the term has no unbound vars, rather than the previous success if no vars at all.

17.8 Chaged lookup and add semantics to correlate with PROLOG. That is, the first axiom or fact added is the first tried. We used to look things up in the reverse order, since that was just a PUSH.

17.9 Allow Trace-BC-Axiom, Untrace-BC-Axiom, Trace-FC-Axiom, and Untrace-FC-Axiom to all accept Forms as arguments, which will cause tracing of all appropriate terms whose LHS or Trigger, respectively, unify with the form to be traced.

A.2 Shortcomings and Enhancements

This section is a guide to the known outstanding problems, and planned future enhancements to Rhet.

A.2.1 Shortcomings

Some of the problems with the current implementation that we know about are:

1. If there is more than one axiom with the same LHS, there is no way for the user to act on only one of them, unless they have different and unique indexes.
2. Only one Rhet window is supported at a time.
3. The index to this document isn't as helpful as it might be. Contributions on missing or useless entries are welcome!

A.2.2 Possible Enhancements

The following are on our enhancement schedule, though it may be a while before this list is exhausted! If there is something on this list you want or need in particular, send mail to Rhet@cs.rochester.edu to request a priority increase. These are presented in no particular priority order.

A.2.2.1 Language Features

New

1. Enhance contexts to allow for matrix (arbitrary DAGs) instead of tree inheritance.
2. Enhance belief ops to allow for additions to MB spaces (requires above)
3. Enhance belief ops to allow for embedded non-tokenized MB, e.g. SBHBMBSB...
4. Complex types (should we figure out how to do them)
5. Handle side effects in builtins (this is hard, since intelligent backtracking doesn't tell us when we are used in the current proof).
6. TMS retriggering retracted facts (find alternate justification) [prefers indexing, int. cache flushing]
7. More TMS stuff, e.g. able to handle rule specialization [requires complex types]
8. Allowing more complex FC triggers, e.g. EQ [requires indexing, compiler, int cache flushing]
9. Meta (i.e. able to calculate what needs done to make something provable or unprovable while keeping integrity of kb (not adding contradictions)) [prefers lisp function interface]

A.2.2.2 User Interface

1. Explanation of Equality Classes (hard, inefficient)
2. Addition of a Stepper
3. Detect Left Recursion (part of caching) [requires goal caching]
4. Better editor interfaces (including incremental compilation)
5. (more) Graphical output
6. On line documentation

A.2.2.3 Performance

1. Goal Caching
2. Intelligent cache flushing [requires goal caching]
3. Better axiom indexing
4. Axiom declarations (allows user to specify even better indexing by declaring vars to be locals, globals, downward, etc.). These can now be declared on **DefRhetPred** forms, but the system doesn't do anything with them.
5. Axiom Compiler [Prefers axiom declarations, requires indexing, int. cache flushing].
6. Incremental hashing [requires indexing]
7. Binary output of compiled code to files [Requires Compiler], and ability to dump references to Rhet terms embedded in lisp expressions to a binary file.
8. Lots of low level stuff should be optimized, i.e. use hashtables, *etc.*. In several cases hashtables are built but not used! In general the rhet-terms, unifier, and interpreter stuff needs cleaned up for efficiency; other levels may not make that much difference on (proof) performance anyway.

Appendix B

Other Specialized Reasoners

One feature of RHET is the ability to extend the system by adding code to reason about certain types of objects to supplement RHET's basic reasoning capabilities. Here are descriptions of such available reasoners:

B.1 TEMPOS

TEMPOS is a time reasoning extension to RHET developed and written by Hans Koomen of TIMELOGIC fame. Basically it is an interface to the TIMELOGIC[Koomen, 1988] system. It is more completely described in [Koomen, 1989], along with examples of it's use. The following is an abbreviated introduction; some text and most examples derive directly from [Koomen, 1989]. Mod

B.1.1 Loading TEMPOS

On the Symbolics, just do:

Rhet-> :Load System Tempos

while on the Explorer, do

Rhet-> (make-system 'tempos)

This will cause the TimeLogic system to be loaded as well.

B.1.2 Lisp Interface

The following functions are available:

Reset-Tempos &key *TT-axioms-p* &rest *TimeLogicKeys*

Initializes the Tempos predicates. Presumably done in a clean state, e.g. after you called (Reset-Rhetorical) If TT-axioms-p, loads a file containing Rhet axioms for reasoning about things holding, occurring or recurring. These are further described in section B.1.4. TimeLogicKeys are passed to the function **Timelogic-Reset-Props**, which is further described in [Koomen, 1988].

New

Trace-Tempos &Optional *Verbose-p*

Turns on tracing of Tempos queries and assertions. If Verbose-p, turns on more Tempos tracing as well as TimeLogic tracing.

Untrace-Tempos

Stops tracing Tempos queries and assertions.

Define-Time &Rest *Terms* &Key *Reference-Time-Term*

Intervals must be declared, but doing so also declares the Rhet type so you don't need to do that explicitly.

E.g.

(B.1) (Define-Time [T1] [T2] ...[Tn] :Reference [Tr])

Mod

defines [T1], [T2], thru [Tn] as time terms, with reference interval [Tr].

B.1.3 Language Interface

New

Note that TEMPOS does pay attention to the Rhet context that assertions and proofs are done in. Unlike Rhet, Tempos, however, does NOT attempt to maintain consistancy between a parent and child contexts. Adding a constraint to a parent context may cause inconsistancies in a child. If completeness is required, a user should make all assertions in all child contexts first, and last make the assertion in the parent.

The following predicates (all builtins) are available:

[Time-Reln Time1 Reln Time2]

Where Reln is either a var or one (or a list of one or more) of the following keywords:

:A (after)

:B (before)

:C (properly contains)

:D (properly during)

:E (equals)

:F (finishes)

:Fi (finished by)

:M (meets)

:Mi (met by)

:O (overlaps)

:Oi (overlapped by)

:S (starts)

:Si (started by)

The predicate succeeds if the known TIMELOGIC relational constraint between Time1 and Time2 is a subset of Reln. The predicate `[Time-Reln]` may be queried with at most 2 out of 3 args unbound; `[Time-Reln]` will backtrack as needed to provide all possible solutions to a query. Note that `[Time-Reln]` is assertable, if it is completely grounded, and succeeds if Reln can consistantly be added as a relational constraint between Time1 and Time2. Mod

New

[Time-Reln-P Time1 Reln Time2]

Like `[Time-Reln]`, but succeeds if Reln *intersects* the known relational constraint between Time1 and Time2. This is used to check if a relation *could* hold, rather than to check if a relationship holds. However, successful return does not indicate that subsequent assertion that `[Time-Reln Time1 Reln Time2]` will succeed. Like `[Time-Reln]`, and two of the tree arguments may be unbound, and the predicate may not be asserted.

[Time-After ?x ?y]
= `[Time-reln ?x :A ?y]`

[Time-Before ?x ?y]
= `[Time-reln ?x :B ?y]`

[Time-Contains ?x ?y]
= `[Time-reln ?x :C ?y]`

[Time-During ?x ?y]
= `[Time-reln ?x :D ?y]`

[Time-Equals ?x ?y]
= `[Time-reln ?x :E ?y]`

[Time-Finishes ?x ?y]
= `[Time-reln ?x :F ?y]`

[Time-Finished-By ?x ?y]
= `[Time-reln ?x :Fi ?y]`

[Time-Meets ?x ?y]
= `[Time-reln ?x :M ?y]`

[Time-Met-By ?x ?y]
= `[Time-reln ?x :Mi ?y]`

[Time-Overlaps ?x ?y]
  = `[Time-reln ?x :O ?y]`

[Time-Overlapped-By ?x ?y]
  = `[Time-reln ?x :Oi ?y]`

[Time-Starts ?x ?y]
  = `[Time-reln ?x :S ?y]`

[Time-Started-By ?x ?y]
  = `[Time-reln ?x :Si ?y]`

[Time-Subint ?x ?y]
  = `[Time-reln ?x '(:D :E :F :S) ?y]`

[Time-Subint! ?x ?y] = `[Time-reln ?x '(:D :F :S) ?y]`

[Time-Disjoint ?x ?y]
  = `[Time-reln ?x '(:A :B :M :Mi) ?y]`

[Time-Disjoint! ?x ?y] = `[Time-reln ?x '(:A :B) ?y]`

[Time-Intersects ?x ?y]
  = `[Time-reln ?x '(:C :D :E :F :Fi :O :Oi :S :Si) ?y]`

[Time-Starts-Later ?x ?y]
  = `[Time-reln ?x '(:A :D :F :Mi :Oi) ?y]`

[Time-Starts-Earlier ?x ?y]
  = `[Time-reln ?x '(:B :C :Fi :M :O) ?y]`

[Time-Finishes-Later ?x ?y]
  = `[Time-reln ?x '(:A :C :Mi :Oi :Si) ?y]`

[Time-Finishes-Earlier ?x ?y]
  = `[Time-reln ?x '(:B :D :M :O :S) ?y]`

All these may be asserted (if fully ground). As queries, one or both intervals may be vars, in which case they succeed if there are bindings satisfying the indicated temporal constraint. They will backtrack as long as there are more appropriate bindings. These predicates are context-sensitive, i.e., you can

(B.2) (Assert-Axioms [Time-Disjoint! Bike-Riding-Time Dinner-Time]
    [WITH "SCENARIO-1"
        [Time-After Bike-Riding-Time Dinner-Time]]
    [WITH "SCENARIO-2"
        [Time-Before Bike-Riding-Time Dinner-Time]]

As the above are related to `[Time-Reln]`, any of the above may take a "-P" suffix as a convenient alternative to `[Time-Reln-P]`.

[Time-Cover Time1 Time2 Time3]

Declares that the interval Time3 covers intervales Time1 and Time2, but the latter two are unconstrained: they may meet, overlap, or be disjoint. It is the same as `[And [Time-Starts Time1 Time3] [Time-Finishes Time2 Time3]]`.

[Time-Cover! Time1 Time2 Time3]

Declares that interval Time3 covers intervals Time1 and Time2, which meet. It is the same as `[And [Time-Meets Time1 Time2] [Time-Cover Time1 Time2 Time3]]`.

[Time-Durn Time1 Durn Time2]

Succeeds if the known durational constraint between Time1 and Time2 is a subset of Durn, a TIMELOGIC durational constraint specification[1]. This is the analogue to `[Time-Reln]`, but with durations. Any two of the three parameters may be unbound. It may be asserted, and succeeds if Durn can consistantly be added as a durational constraint between Time1 and Time2.

[Time-Durn-P Time1 Durn Time2]

`[Time-Durn-P]` is to `[Time-Durn]` as `[Time-Reln-P]` is to `[Time-Reln]`. It checks that Durn *intetersects* with the known durational constaint specification between Time1 and Time2, and succees, again, does not guarentee that assertion of `[Time-Durn Time1 Durn Time2]` will necessarily succeed. Any two of the three arguments may be unbound. The predicate may not be asserted.

[Time-= ?x ?y]

= `[Time-Durn ?x 1 ?y]`

[Time-< ?x ?y]

= `[Time-Durn ?x (0 (1)) ?y]`

[Time-> ?x ?y]

= `[Time-Durn ?x ((1) :INF) ?y]`

[Time-<= ?x ?y]

= `[Time-Durn ?x (0 1) ?y]`

[Time->= ?x ?y]

= `[Time-Durn ?x (1 :INF) ?y]`

[Time-<> ?x ?y]

= `[Time-Durn ?x (OR (0 (1)) ((1) :INF) ?y]`

All these may be asserted (if fully ground). As before, one or both intervals may be vars as a query, and succeed if there are bindings satisfying the indicated durational constraint, backtracking so long as there are other appropriate bindings. They are context sensitive.

As the above are related to `[Time-Durn]`, any of the above may take a "-P" suffix as a convenient alternative to `[Time-Durn-P]`.

---

[1]A durational constraint specifies relative magnituedes between intervals; these are described in more detail in [Koomen, 1988].

[Time-Skolem ?x *&Optional* Reference-Time]

Fails if ?x is bound, otherwise succeeds binding ?x to a skolem interval and installing all posted temporal constraints on ?x. If backtracked, uninstalls all posted temporal constraints and fails. *E.g.*, `[AND [POST [Time-Before ?x SUNDAY]] [Time-Skolem ?x]]` is the same as `[AND [Time-Skolem ?x] [ASSERT-FACT [Time-Before ?x SUNDAY]]]`

Can not be asserted.

New

If Reference Time is supplied, it must be a defined time term, and becomes the reference time for the skolem. Note that without the optional Reference-Time argument, the builtin is redundant with the Rhet builtin `[Skolemize]`.

A query on a grounded Tempos base term (relating two intervals) causes the relation to be asserted if either interval is a Skolem, and it is possible to do so, backtracking if necessary.

New

B.1.4 Truth in Time Axioms

These predicates are optionally added (see **Reset-Tempos**) to allow reasoning about terms that are true relative to a time interval. Note that since currently Rhet does NOT allow one to use a predicate as an argument to another function, one must use a function term (an object which is a subtype of type `*T-U` rather than `*T-Fact`) instead. This can be directly asserted by using a predicator, such as "TRUE", *e.g.*one could have a function term `[Mother-Of Sam Bill]`, but to assert it as true, one would assert `[True [Mother-Of Sam Bill]]`. This sort of thing is only needed when one wants to both use a term as a component in another (as these builtins require), and deal with the object as either "True" or "False".

[TT Time1 Term1]

True Throughout: Term1 is true over interval Time1. In general it is preferable to query truth in terms of `[TT]`, but to assert it in terms of `[MT]`. Note that `[TT Time1 Term1]` is true if `[TT Time2 Term1]` is true, Term1 is declared to be of type *liquid, downward hereditary, or concatenable*, and Time1 is appropriately related to Time2.

[MT Time1 Term1]

Maximally True: Term1 is true over interval Time1, and *not* true over any interval that overlaps or is overlapped by Time1. That is, any interval ?i*T-Time over which Term1 is true must be euql, contained in or disjoint from Time1.

[FT Time1 Term1]

False Throughout: Term1 is false over interval Time1. Note that this is distinct from `[Not [TT Time1 Term1]]`!

[MF Time1 Term1]

Maximally False: Term1 is false over interval Time1, such that any other itnerval over which Term1 is false is either within or disjoint from Time1. Note that this is distinct from `[NOT [MT Time1 Term1]]`!

[RT Time1 Term1]
Repeatedly True: Term1 is repeatedly true over interval Time1, that is, there exist a series of intervals $?j_1$*T-Time...$?j_n$*T-Time, each $?j_i$ meeting $?j_i + 1$, the first starting Time1 and the last finishing Time1, such that Term1 is maximally true over each $?j_i$.

[Time-Upward Term1]
Term1 is true over interval ?i*T-Time if Term1 is true over all intervals ?j*T-Time contained in ?i*T-Time.

[Time-Downward Term1]
If Term1 is true over interval ?i*T-Time then Term1 is true over any interval contained in ?i*T-Time.

[Time-Liquid Term1]
= `[And [Time-Upward Term1] [Time-Downward Term1]]`.

[Time-Concatenable Term1]
If Term1 is true over intervals ?i*T-Time and ?j*T-Time, and ?i and ?j meet, then Term1 is true over interval ?k*T-Time, where ?k is the cover of ?i and ?j.

For example,

```
Rhet-> (Define-Time [18-Jan-89] [Jan-89])

Rhet->

(Assert-Axioms [TT Jan-89 Winter]
               [Time-Liquid Winter]
               [Time-During 18-Jan-89 Jan-89])


Rhet-> (Prove [TT 18-Jan-89 Winter])
[TT 18-Jan-89 Winter]
Rhet->
```

[Time-Mutex Time1 Time2]
If true, any occurrences of Time1 and Time2 are said to be mutually exclusive (forced to be temporally disjoint).

For additional material on these axioms, including when they are provable, see [Koomen, 1989].

Bibliography

[Allen and Miller, 1986] James F. Allen and Bradford W. Miller, "The HORNE Reasoning System in COMMON LISP," Technical Report 126, University of Rochester, Computer Science Department, August 1986, Revised.

[Allen and Miller, 1990] James F. Allen and Bradford W. Miller, "The Rhetorical Knowledge Representation System: A Tutorial Introduction," Technical Report 325, University of Rochester, Computer Science Department, December 1990.

[Koomen, 1988] Johannes A.G.M. Koomen, "The TIMELOGIC Temporal Reasoning System," Technical Report 231 (revised), University of Rochester, Computer Science Department, October 1988.

[Koomen, 1989] Johannes A.G.M. Koomen, *Reasoning About Recurrence*, PhD thesis, University of Rochester, July 1989, Also TR 307.

[Kornfeld, 1983] W. A. Kornfeld, "Equality for Prolog," In *Proceedings, 8th IJCAI*, Karlsruhe, W. Germany, August 1983.

[Miller, 1990a] Bradford W. Miller, "The RHET Plan Recognition System," Technical Report 298, University of Rochester, Computer Science Department, January 1990.

[Miller, 1990b] Bradford W. Miller, "Rhet Programmer's Guide," Technical Report 239 (revised*), University of Rochester, Computer Science Department, December 1990.

[Steele Jr., 1990] Guy L. Steele Jr., *Common Lisp the Language 2/e*, Digital Press, 1990, ISBN 1-55558-042-4.

Index